Gonçalo Manuel Conduto Rodrigues

5º ano do Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Protocolo MAC para redes com nós sensores móveis e nós de recolha móveis e sensores SunSPOT

Dissertação para obtenção do Grau de Mestre em Engenharia Electrotécnica e de Computadores

Orientador: Luís Bernardo, Professor Auxiliar, FCT

Presidente: Prof. Doutor Paulo da Fonseca Pinto
Arguente: Prof. Doutor Paulo José Carrilho de Sousa Gil

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro de 2011

Protocolo MAC para redes com nós sensores móveis e nós de recolha móveis e sensores SunSPOT

Resumo

As redes de sensores sem fios móveis têm requisitos bastante diferentes, em termos de protocolos de controlo de acesso ao meio (MAC), dos outros tipos de redes. Os protocolos destas redes devem apresentar um baixo consumo energético e ser capazes de lidar tanto com grandes períodos sem tráfego como com períodos de tráfego muito intenso.

O protocolo MMH-MAC (*Mobile Multimode Hybrid - MAC*) foi desenhado para lidar com estas situações e neste trabalho propõe-se a implementação deste protocolo. O protocolo apresenta um funcionamento assíncrono para alturas em que o tráfego na rede é baixo e um modo síncrono quando o tráfego na rede é alto. Isto permite optimizar o consumo de energia. Nesta dissertação foi realizada uma realização do modo assíncrono do protocolo MMH-MAC.

Para o desenvolvimento deste protocolo foi usada a plataforma SunSPOT. Esta plataforma é completamente implementada em Java, o que apresenta vantagens em termos de facilidade de desenvolvimento de aplicações. No entanto, o desenvolvimento do protocolo MAC revelou-se complicado.

Foram criados cenários para a análise do desempenho do protocolo e este foi comparado com uma implementação do mesmo protocolo usando uma plataforma diferente. É também feita uma análise à própria plataforma.

Palavras-Chave

Redes de Sensores Sem fios Móveis, Protocolo de Controlo de Acesso ao Meio, Plataforma SunSPOT.

Abstract

The mobile wireless sensor networks have different requirements in terms of medium access control (MAC) protocols when compared to other types of networks. Protocols for these networks must present low energy consumption and still be able to handle very low traffic periods and peaks of very high traffic.

The MMH-MAC (*Mobile Multimode Hybrid – MAC*) protocol was designed with this in mind. The protocol presents an asynchronous mode for periods of low traffic and a synchronous mode for high traffic. This allows for the optimization of energy savings. The asynchronous mode of MMH-MAC is implemented in this thesis.

The SunSPOT platform was used for the development of this protocol. This platform is completely implemented in Java, which brings advantages in terms of ease of application development. However, MAC protocol implementation is far more complex.

The performance of the protocol was analyzed and then compared to the performance of the same protocol using a different development platform. An analysis is also made on the platform itself.

Keywords

Mobile Wireless Sensor Networks, Medium Access Control Protocol, SunSPOT Development Platform.

Índice de Matérias

Capítulo 1. Introdução 1
1.1. Introdução 1
1.2. Estrutura da Dissertação 2
Capítulo 2. Trabalho Relacionado 3
2.1. Redes de Sensores Sem Fios 3
2.1.1. Diferentes Tipos de Redes 3
2.1.2. Serviços da Rede 4
2.1.3. Protocolos de Comunicação 6
2.2. Protocolos de Controlo de Acesso ao Meio 7
2.2.1. Protocolos de Controlo de Acesso ao Meio Tradicionais 7
2.2.2. Principais Causas de Consumo de Energia 9
2.2.3. Principais Soluções 10
2.2.4. Protocolos de Escalonamento 11
2.2.5. Protocolos com Período Ativo Comum 12
2.2.6. Protocolos com Preâmbulos 14
2.2.7. Protocolos Híbridos 15
Capítulo 3. Sistema Proposto 17
3.1. Descrição formal do protocolo realizado 18
3.2. AnyMAC 20
3.2.1. Características do AnyMAC 20
3.2.2. Arquitetura dos SunSPOT 21
3.2.3. Arquitetura do AnyMAC 24
3.3. Implementação do MH-MAC em AnyMAC 32
Capítulo 4. Análise de Desempenho 36
4.1. Ambiente de Teste 36
4.2. Medições Realizadas 37
4.3. Resultados 38
4.3.1. Temperatura 38
4.3.2. Estado do sensor sem fios 39
4.3.3. Bateria 41
4.4. Comparação com outros Sensores 42
Capítulo 5. Conclusões 43
5.1. Síntese Geral 43
5.2. Conclusões 44
5.3. Trabalho Futuro 45
Capítulo 6. Bibliografia 47

Apêndice A. MHMAC.java 49
Apêndice B. MHMACAddress.java 61
Apêndice C. MHMACAddressTranslator.java 63
Apêndice D. MHMACPacket.java 65
Apêndice E. MHMACPacketFactory.java 69
Apêndice F. MHMACPacketReceiver.java 71
Apêndice G. MHMACPacketReceptionListener.java 75
Apêndice H. MHMACPacketSender.java 77
Apêndice I. MHMACPacketTransmissionListener.java 81
Apêndice J. MHMACSettingsManager.java 83
Apêndice K. DataCollectorUtils.java 85
Apêndice L. SPOTDataSender.java 89
Apêndice M. CollectDataHost.java 91

Índice de Figuras

Figura 3.1 – Transmissão de dados assíncrona em *unicast* 18
Figura 3.2 – Transmissão assíncrona de dados em *broadcast* 19
Figura 3.3 – Arquitetura dos sensores sem fio SunSPOT 21
Figura 3.4 – Arquitetura do AnyMAC 24
Figura 3.5 – Formato dos Pacotes 25
Figura 3.6 – Arquitectura dos Pacotes 25
Figura 3.7 – Arquitetura da *Hardware Abstraction Layer* 27
Figura 3.8 – Arquitetura do MAC 29
Figura 3.9 – InformationDatabase 31
Figura 4.1 – Ambiente de teste 36
Figura 4.2 – Gráfico da Temperatura medida ao longo do Tempo 38
Figura 4.3 – Gráfico dos Estados do Sensor 39
Figura 4.4 – Gráfico dos Estados do Sensor para a implemantação padão 40
Figura 4.5 – Gráfico da Bateria ao longo do Tempo 41

Índice de Tabelas

Tabela 4.1 - Comparação dos consumos 42

Acrónimos

ACK – Acknowledge

AODV – Ad hoc On Demand Distance Vector

API – Application Programming Interface

B-MAC – Berkeley - Medium Access Control

CCA – Clear Access Channel

CSMA – Carrier Sense Multiple Access

CSMA/CA – Carrier Sense Multiple Access/Collision Avoidance

CTS – Clear To Send

MAC – Medium Access Control

MH-MAC – Multimode Hybrid - Medium Access Control

MMH-MAC – Mobile Multimode Hybrid - Medium Access Control

MS-MAC – Medium Access Protocol for Sensor Networks

LPL – Low Power Listening

PACK – Preamble Acknowledge

RTS – Request To Send

S-MAC – Sensor - Medium Access Control

T-MAC – Timeout - Medium Access Control

TDMA – Time Division Multiple Access

Z-MAC – Zebra Medium Access Control

Capítulo 1. Introdução

1.1. Introdução

Um dos principais modos de desenvolvimento de protocolos para sensores sem fios foi até há pouco tempo a utilização de TinyOS e muitos dos protocolos desenvolvidos entravam em consideração com várias limitações existentes em TinyOS. Com o aparecimento da plataforma de sensores sem fios SunSPOT, que apresentam maiores recursos computacionais e têm uma maior facilidade de programação de aplicações (Java) embora menos possibilidades de acesso às interfaces de baixo nível do hardware, era importante reanalisar os protocolos existentes e observar se são realizáveis com sensores SunSPOT.

Um dos protocolos de controlo de acesso ao meio (MAC) desenvolvidos para TinyOS foi o MMH-MAC (*Mobile Multimode Hybrid* - MAC). Este protocolo foi originalmente desenvolvido para os sensores TelosB, que usam o mesmo integrado de rádio que os sensores SunSPOT (CC2420).

Com este trabalho pretende-se desenvolver uma reimplementação do protocolo usando como base a plataforma de sensores sem fios SunSPOT. Este protocolo foi desenhado tendo em consideração o sistema operativo TinyOS, e um dos objectivos foi comparar a complexidade e o desempenho das duas implementações.

1.2. Estrutura da Dissertação

A dissertação encontra-se organizada em cinco capítulos, conforme se resume em seguida.

No Capítulo 2 efetua-se uma análise dos protocolos de nível MAC existentes para redes de sensores sem fios. Em cada ponto efetua-se uma análise critica acerca desses protocolos. Analisam-se também as redes de sensores sem fios, em particular o seu tipo, os serviços que providenciam e os vários níveis onde os protocolos de rede atuam.

No Capítulo 3 é descrito o modelo de protocolo que foi implementado. É também descrito o ambiente onde o protocolo foi implementado, os problemas encontrados e as soluções usadas na sua resolução.

No Capítulo 4 procede-se à descrição do cenário usado na análise do desempenho do protocolo, assim como do conjunto de leituras de sensores e outra medições realizadas. Segue-se uma análise dos resultados.

No Capítulo 5 é feita uma análise geral do trabalho realizado assim como as contribuições deste trabalho e as questões em aberto para trabalho futuro.

Capítulo 2. Trabalho Relacionado

Tem havido, nos últimos anos, uma grande investigação na área das redes de sensores sem fios. Neste capítulo, descrevem-se os protocolos de acesso ao meio tradicionais e suas principais limitações numa rede sem fios. Depois apresentam-se algumas soluções para as deficiências apresentadas e também exemplos de protocolos existentes que as usam.

São também apresentados os vários tipos de redes de sensores sem fios existentes e os serviços desenvolvidos para a manutenção da rede. Por fim, descrevem-se brevemente os vários níveis de rede existentes e sua função.

2.1. Redes de Sensores Sem Fios

As redes de sensores sem fios ganharam nos últimos tempos grande importância [1]. Os sensores constituintes destas redes são pequenos com capacidade de processamento limitada e são pouco dispendiosos quando comparados com os sensores tradicionais. Visto estes sensores terem recursos limitados e serem geralmente colocados em locais de difícil acesso é usado um rádio para a comunicação.

Tipicamente as redes são compostas por um número variado de sensores que pode ir desde as dezenas até aos milhares. Estas redes têm muitas aplicações práticas tais como: rastreamento de alvos, vigilância, ajuda em desastres naturais, monitores biomédicos e exploração de ambientes hostis para nomear alguns.

As redes de sensores sem fios têm um desenho e restrições próprias diferentes das redes tradicionais. As restrições incluem baixa largura de banda, curto alcance nas comunicações, energia limitada, e baixo poder de processamento e armazenamento em cada sensor. Alguns destes limites são impostos pelo ambiente onde a rede é disposta. A pesquisa desenvolvida nesta área pretende minimizar as restrições descritas acima introduzindo novos conceitos, criando ou melhorando protocolos existentes, criando novas aplicações e desenvolvendo novos algoritmos.

2.1.1. Diferentes Tipos de Redes

Dependendo do local onde são instaladas, podem-se definir vários tipos de redes de sensores sem fios.

- Terrestre

  É uma rede composta por centenas ou milhares de nós. Os seus principais desafios são a agregação dos dados para minimizar o gasto de energia e reduzir o atraso na recepção dos dados, encontrar a rota ideal para transmissão dos dados, eliminar a redundância e manter a conectividade da rede.

Algumas das aplicações possíveis deste tipo de redes são a de monitorização de ambientes industriais e a de exploração da superfície terrestre.

- Debaixo de Terra

É uma rede composta por sensores colocados em grutas ou minas e tem como principais desafios os grandes níveis a atenuação e perda de sinal nas transmissões.

Algumas das aplicações possíveis deste tipo de redes são a de monitorização de estruturas, monitorização do ambiente tal como água, solo ou minerais e monitorização da agricultura.

- Debaixo de Água

É uma rede composta por sensores ou veículos colocados de baixo de água e tem como principais desafios as falhas de sensores devido à corrosão, a largura de banda utilizável e a grande atraso nas transmissões.

Algumas das aplicações possíveis deste tipo de redes são a de monitorização da poluição e a de monitorização sísmica.

- Multimédia

É uma rede composta por sensores capazes de processar dados multimédia. A grande largura de banda necessária e os altos consumos de energia são os principais desafios desta rede.

Uma das aplicações deste tipo de redes é a de melhorar as aplicações de monitorização e rastreamento.

- Móvel

É uma rede composta por sensores com a capacidade de se moverem, ou assentes em objectos ou seres móveis.

Algumas das aplicações possíveis deste tipo de redes são a de monitorização ambiental, monitorização de habitats e rastreamento de alvos.

2.1.2. Serviços da Rede

Nas redes de sensores sem fios foram desenvolvidos vários serviços que facilitam a realização de aplicações e contribuem para melhorar o seu desempenho [2]. Seguidamente segue-se uma descrição de alguns serviços relevantes.

- Localização

Os sensores são geralmente colocados aleatoriamente numa rede sem prévio conhecimento da sua localização. Entre os métodos para obter a localização está o de dotar os sensores com um GPS, mas este método apresenta problemas quando o ambiente não permite o uso do GPS e também devido ao elevado consumo energético da localização GPS. Para permitir a localização, basta colocar alguns sensores que conhecem a sua posição espalhados entre os restantes. Podem-se utilizar outros métodos [3], como a estimação do ângulo, da distância, da diferença entre tempos de chegada, ou mesmo só a informação sobre a vizinhança. Alguns destes métodos apresentam problemas de escalabilidade com o crescimento da rede, pois exigem densidades elevadas de nós com conhecimento da localização

- ## Sincronização

A sincronização temporal nas redes é importante pois permite aos sensores usarem protocolos com escalonamento de transmissão de dados. Isto reduz as colisões e as retransmissões e aumenta a poupança de energia. Todos os protocolos de sincronização de tempo tentam estimar a incerteza e sincronizar os relógios locais com a rede.

- ## Cobertura

É importante calcular a cobertura de uma rede para uma determinada área quando se avalia a sua eficiência. A pesquisa existente foca a sua atenção no contexto da redução de energia. As principais técnicas propostas focam a sua atenção ou em selecionar o número mínimo de sensores que têm que estar ativos para manter a cobertura, ou em estratégias de colocação dos sensores.

- ## Compressão e Agregação

Tanto a compressão como a agregação dos dados reduz o custo de comunicação entre os sensores. Ambas as técnicas são necessárias para lidar com grandes quantidades de dados a transmitir. Nas técnicas de compressão dos dados, estes são comprimidos nos sensores antes de serem enviados e só são descomprimidos no destino final. Nas técnicas de agregação, os dados são recolhidos de vários sensores, agregados e só depois enviados para o destino final. Ambas as técnicas tentam resolver os problemas de consumo de energia, de fiabilidade, escalabilidade e eficiência.

- ## Segurança

Uma rede sem fios é vulnerável a ataques. Um inimigo pode comprometer um ou mais sensores, alterar a integridade dos dados, observar os dados transferidos, injetar dados falsos e esgotar os recursos da rede. Existem várias limitações em incorporar segurança numa WSN, tais como os limites de memória e computação dos sensores e de débito e

energéticos na comunicação. A criação de protocolos de segurança nestas condições requere um profundo conhecimento das limitações existentes e conseguir atingir um desempenho aceitável para a aplicação em causa.

2.1.3. Protocolos de Comunicação

Nas redes de sensores sem fios foram desenvolvidos vários protocolos de comunicação com caraterísticas especificas [2], sumariados de seguida.

- #### Nível de Transporte

O nível de transporte assegura a qualidade e a confiança dos dados no receptor e no emissor. Os protocolos do nível de transporte devem providenciar variados níveis de confiança para diferentes aplicações. Devem também suportar a recuperação de pacotes perdidos, ou entre cada troço de rede ou entre o emissor e o receptor. Além disso, os protocolos devem também suportar mecanismos para controlo de congestão.

- #### Nível de Rede

O nível de rede lida com o encaminhamento dos pacotes através da rede, desde o emissor até ao receptor. Os protocolos de encaminhamento de rede das WSNs diferem dos protocolos tradicionais em vários aspectos. Um deles é a inexistência de endereços IP de âmbito global nos sensores; logo o encaminhamento baseado em IP não pode ser usado diretamente. Nas redes de sensores sem fios é usado vulgarmente usado o encaminhamento orientado para dados (*data driven*), embora também exista a possibilidade de realizar a associação de sensores a endereços IP utilizando 6lowPAN [4]. Estes protocolos têm que ser escaláveis, capazes de manter a comunicação entre muitos sensores e respeitar as limitações impostas por estes tipos de redes.

- #### Nível de Dados

A grande preocupação do nível de dados é a transferência de dados entre dois extremos de uma ligação. Como a rede é sem fios, o controlo de acesso ao meio toma especial importância. Os protocolos de acesso ao meio devem ter, entre outros, os seguintes atributos: ser energeticamente eficientes, capazes de escalar com o número de nós, controlo de fluxo e controlo de erros. Além de terem de providenciar os atributos mencionados acima, os protocolos estão também sujeitos a várias limitações tais como topologia, mudanças na rede e energia limitada.

- Nível Físico

O nível físico providencia uma interface para a transmissão de bits através do meio e interage com o nível de dados para a transmissão e recepção dos mesmos. Esta interação é muito importante pois em meios sem fios o meio tem um número de erros elevado.

É neste nível que é controlado o rádio, o que tem grandes impactos na minimização de energia.

- Interações entre Níveis

As aproximações em que os vários níveis são tratados como um todo são mais eficientes que as aproximações tradicionais. Nas aproximações tradicionais os vários níveis são tratados separadamente fazendo com que haja mais *overhead*. Nas outras a informação é partilhada entre os vários níveis e elas são optimizadas como um todo, sendo possível atingir reduções de energia não possíveis com as aproximações tradicionais.

2.2. Protocolos de Controlo de Acesso ao Meio

As comunicações sem fios são inerentemente por difusão e estão sujeitas a interferências, fazendo com que os seus protocolos de controlo de acesso ao meio (MAC) tenham um papel relevante na sua eficiência. Isto é particularmente verdade para as redes de sensores sem fios constituídas por grandes quantidades pequenos sensores com autonomia muito limitada e requisitos de operação durante anos sem intervenção humana.

Tem havido um crescente interesse no desenvolvimento e optimização dos protocolos de controlo de acesso ao meio, com o principal intuito de reduzir o seu consumo de energia oferecendo nalguns casos latências limitadas ou débitos controlados.

Seguidamente são apresentadas as principais abordagens usadas nos protocolos de controlo de acesso ao meio e seus problemas. São também apresentadas as principais causas do consumo de energia nos nós e soluções para as reduzir. Posteriormente são analisados vários protocolos que tentam colmatar os problemas apresentados.

2.2.1. Protocolos de Controlo de Acesso ao Meio Tradicionais

Existem duas aproximações principais para a regulação do acesso ao meio: métodos baseados na contenção ou métodos baseados na reserva do meio. Assim sendo, qualquer protocolo derivado será baseado numa destas aproximações ou uma combinação das duas.

- ## Protocolos Baseados na Reserva do Meio

Esta aproximação necessita do conhecimento da topologia da rede para estabelecer um escalonamento que permita que cada nó possa aceder ao meio e comunicar com os outros nós. O escalonamento pode ter vários objectivos tais como: reduzir colisões, assegurar que nenhum nó é preterido em termos de acesso, etc.

TDMA (*Time Division Multiple Access*) é um método de acesso representante das aproximações baseadas na reserva do meio. Esta aproximação consiste na divisão do tempo em *frames* (tramas) que por sua vez são divididas em *slots*. Durante uma *frame* cada nó tem um *slot* associado, no qual apenas ele tem o direito de transmitir, evitando assim as colisões e aumentando o débito (*throughput*) em redes com muito tráfego. Esta aproximação também assegura que todos os sensores têm a mesma oportunidade para transmitir dados, pois cada nó está associado a um *slot* em cada *frame*.

Apesar de apelativa, esta aproximação tem as suas desvantagens resultantes de ser necessário o conhecimento da topologia, da sincronização temporal dos nós e da coordenação no acesso aos *slots*.

- ## Protocolos Baseados na Contenção do Meio

Esta aproximação é bastante mais simples quando comparada com as de reserva do meio, visto não ser necessário o conhecimento da topologia nem a sincronização global dos nós. Nesta aproximação, os nós competem pelo acesso ao meio e apenas ao vencedor é permitido transmitir.

ALOHA e CSMA (*Carrier Sense Multiple Access*) são os grandes representantes desta aproximação.

Em CSMA um nó que queira transmitir tem que primeiro amostrar o meio para ver se este se encontra ocupado. Caso o nó encontre o meio ocupado, adia a sua transmissão para não interferir com a transmissão em curso. Por outro lado, se o meio não se encontrar ocupado, o nó começa a transmitir. Como a única condicionante para a transmissão é o estado do meio, o CSMA é um bom candidato para redes dinâmicas e com nós móveis.

Apesar do seu sucesso os protocolos de contenção do meio sofrem de uma degradação no desempenho, com a diminuição do débito e com o aumento de tráfego na rede. Além disso, visto não ser usada mais nenhuma informação além do estado do meio, esta aproximação não consegue atingir os mesmos níveis de desempenho das aproximações com reserva do meio.

2.2.2. Principais Causas de Consumo de Energia

Seguidamente são apresentadas as principais causas do consumo de energia.

- Colisões

Pode acontecer que um nó esteja na área de recepção de dois ou mais nós que estão a transmitir. Quando isto acontece, o nó não consegue receber a informação e a energia gasta na transmissão e recepção de pacotes durante uma colisão é desperdiçada. Devido a este problema ter um grande impacto no desempenho dos protocolos, os protocolos devem apresentar uma solução para minimizar ou eliminar este problema.

- *Overhearing*

O *overhearing* acontece quando um nó desperdiça energia a receber pacotes irrelevantes. Exemplos de pacotes irrelevantes são pacotes *unicast* dirigidos a outro nó ou pacotes de *broadcast* redundantes.

- *Overhead*

O *overhead* do protocolo pode resultar em perdas de energia a quando do envio e recepção de pacotes de controlo. Por exemplo os pacotes de controlo RTS (*Request To Send*) e CTS (*Clear To Send*) usados em alguns protocolos não contêm dados úteis mas o seu envio e recepção consome energia (embora evite o problema do nó escondido).

- *Idle Listening*

O *idle listening* acontece quando um nó não sabe quando vai receber um pacote, que é o caso mais comum. Neste caso, o nó mantém o rádio ligado à espera que chegue algum pacote. A energia desperdiçada nestes casos é considerável, mesmo quando o nó não está a receber ou a enviar.

Quando as aplicações usadas nas redes de sensores geram pouco tráfego na rede é esperado que o meio esteja quase sempre disponível. Nestas condições o *Idle Listening* é a principal fonte de gastos de energia. Sem qualquer controlo, os nós, gastam enormes quantidades de energia mantendo o rádio ligado desnecessariamente. Devido a isso é muito importante que os protocolos de controlo de acesso ao meio ponham os nós a dormir (i.e. num modo de poupança de energia em que se desliga o rádio) durante a maioria do tempo.

2.2.3. Principais Soluções

Apesar de terem existido extensos esforços de pesquisa na optimização das aproximações descritas acima, tal como a criação de soluções híbridas, estes protocolos não são necessariamente os mais adequados para as redes se sensores sem fios.

Tal como descrito na secção 2.1.1 as redes de sensores sem fios têm limitações energéticas muito elevadas e funcionam com pouco tráfego. São descritas algumas aproximações que tendem a colmatar as deficiências apresentadas, na secção 2.2.2.

- #### Redução de Colisões

Enquanto as colisões são naturalmente eliminadas nos protocolos baseados na reserva do meio, os protocolos baseados na contenção têm que prestar especial atenção a este aspecto.

A técnica CSMA/CA (CSMA/*Collisions Avoidance*) é das mais usadas na redução de colisões. Esta técnica é baseada na troca de pacotes de dimensão reduzida, os pacotes RTS e CTS. Apesar desta troca de pacotes reduzir eficazmente as colisões nas redes sem fios tradicionais, ela tem alguns problemas nas redes de sensores sem fios. Primeiro, os pacotes de dados das redes de sensores são também eles de dimensão reduzida, fazendo com que a probabilidade de colisão seja da mesma ordem de grandeza da dos pacotes RTS/CTS. Segundo, a troca destes pacotes faz aumentar o consumo de energia; e por último, esta troca de pacotes só pode ser usada em transmissões *unicast*.

A técnica MACA (*Multiple Access Collision Avoidance*) melhora a técnica CSMA/CA adicionando um tempo aleatório de espera antes de enviar um pacote RTS para evitar colisões resultantes do reenvio de mensagens por vários nós vizinhos.

Enquanto as técnicas apresentadas anteriormente, CSMA/CA e MACA, tentam reduzir as colisões de forma igual para todos os pacotes, em certas aplicações existem pacotes que são mais importantes que outros. A técnica Sift [5] tenta resolver o problema apenas reduzindo as colisões dos primeiros R pacotes de N possíveis.

- #### Redução de *Overhead*

A técnica CSMA pode ser ainda optimizada em termos de *overhead* (sobrecusto de comunicação). CSMA/ARC (*Adaptative Rate Control*) [6] é uma técnica que modifica o CSMA/CA omitindo a troca de pacotes RTS/CTS e aplicando um tempo de espera que é regulado conforme a periodicidade da aplicação. Esta técnica reduz ainda mais o custo de comunicação evitando o uso de pacotes ACK (*Acknowledge*), uma vez que o reenvio dos pacotes para outro nó realiza implicitamente essa confirmação.

- Redução de *Overhearing*

O *overhearing* (escuta de pacotes não destinados ao nó) pode ser evitado através de uma filtragem de pacotes baseada no endereço de destino dos pacotes.

A técnica PAMAS (*Power Aware Multi-Access with Signalling*) [7] é baseada em MACA mas usa um canal diferente para a troca de pacotes RTS/CTS. PAMAS usa esta troca de pacotes para informar os nós sobre o destino a origem e a duração da transmissão, fazendo com que um nó que não esteja envolvido na transmissão possa dormir para não receber pacotes que não lhe são destinados. Esta técnica foi desenvolvida para redes *ad-hoc* com muito tráfego, onde o número de transmissões é grande. Em redes de sensores a situação é diferente - o tráfego é baixo e a troca de pacotes RTS/CTS é muitas vezes omitida, fazendo com que esta técnica não poupe muita energia.

- Redução de *Idle Listening*

A ideia fundamental em redes sem fios é de por os nós a dormir o máximo tempo possível e ao mesmo tempo de evitar a perda de pacotes por estes se encontrarem a dormir e de reduzir o *overhearing* e o *overhead*. Em redes com uma estrutura pré-existente tais como WLANs, a poupança de energia através da redução do *Idle Listening* (escuta do canal vazio) não é tão difícil de ser alcançada, pois a infraestrutura pode ser alimentada continuamente. Isto faz com que seja possível colocar os nós em dois modos: a dormir ou ativos. O IEEE 802.11 PSM (*Power Save Mode*) para BSS (*Basic Service Set*) é um exemplo desta técnica.

Esta técnica é usada em WLANs onde existe um ponto de acesso e todos os nós são capazes de comunicar diretamente uns com os outros. Cada nó que quer poupar energia envia um pacote para o ponto de acesso. Quando é recebida uma resposta positiva, o nó começa o procedimento PSM - ele adormece, apenas acordando periodicamente para receber informação do ponto de acesso. Visto um nó neste estado não poder receber pacotes usando os procedimentos tradicionais, o ponto de acesso armazena todos os pacotes destinados a este nó. Aquando da sua comunicação com o nó, o ponto de acesso informa-o de que tem pacotes para ele. Se os pacotes forem de *broadcast*, é o ponto de acesso que determina quando é que estes vão ser enviados e o nó pode decidir recebê-los ou não. Se os pacotes forem *unicast* é o nó que decide quando é que os quer receber, enviando um pacote para o ponto de acesso informando-o que está pronto a receber. Após a recepção deste pacote, o ponto de acesso começa a enviar os pacotes previamente guardados para o nó.

2.2.4. Protocolos de Escalonamento

Tráfego periódico e grande quantidade de transferências são melhor suportadas por aproximações baseadas em reserva do meio. Geralmente, no contexto de redes de sensores

sem fios, são usadas aproximações baseadas em TDMA (*Time Division Multiple Access*) e FDMA (*Frequency Division Multiple Access*) onde diferentes *slots* e frequências podem ser usados por sensores diferentes. TDMA é atraente porque depois de estabelecido o escalonamento não há colisões, *overhearing* e os períodos de *idle listening* são minimizados.

- TRAMA

O protocolo TRAMA (*TRaffic Adaptive Medium Access*) [8] determina um escalonamento sem colisões e cria e atribui as ligações de acordo com o tráfego esperado. O protocolo é composto por duas fases: a da criação da topologia local e a do envio e recepção de dados. O principal problema com este protocolo é a sua complexidade e por assumir que os sensores estão sincronizados em toda a rede.

- FLAMA

O protocolo FLAMA (*FLow-Aware Medium Access*) [9] melhora o protocolo TRAMA, eliminando a troca de informação periódica com vizinhos a dois saltos de distância.

- µMAC

O protocolo µMAC (micro MAC) [10] usa uma ideia similar ao TRAMA, mas com algumas diferenças. Primeiro, a sincronização dos nós é obtida através do uso de um sinal externo. Segundo, também constrói a topologia para nós a dois saltos de distância e usa contenção para comunicação. Em terceiro, o próprio período de contenção é dividido em *slots* e os nós usam esses *slots* para transmitir. Por último, não existe informação se os pacotes são bem recebidos ou não.

2.2.5. Protocolos com Período Ativo Comum

Os seguintes protocolos definem períodos comuns em que os nós ou estão ativos ou estão a dormir. Esta aproximação requere que os nós mantenham um nível elevado de sincronização, para assegurar que existem períodos comuns em todos os nós. Assim, é possível usar os períodos ativos para comunicação entre os nós e os outros para poupança de energia. Durante os períodos ativos, os nós usam aproximações de contenção do meio para transmitir.

- S-MAC

O protocolo S-MAC (*Sensor Medium Access Control*) [11] lida com o *idle listening* fazendo os nós alternar entre dois estados: um estado onde estão acordados e um estado onde estão a dormir. Os períodos em que o nó está acordado têm um tamanho fixo enquanto os períodos onde está a dormir depende do parâmetro do *duty-cycle*, associado à percentagem do tempo em que o nó está ativo. O protocolo divide os períodos ativos em dois subperíodos: um para

troca de pacotes SYNC e outro para troca de pacotes de dados. Cada subperíodo é por sua vez dividido em *slots*. Em ambos os subperíodos, os nós analisam o meio e se este estiver livre transmitem no *slot* seguinte.

Cada nó tem o seu escalonamento de periodos a dormir, com o qual determina quando é que alterna de estado. No seu estado inicial, o nó escuta o meio para ver se já existe algum nó na rede que esteja a disseminar o escalonamento. Se algum escalonamento for encontrado, então este é adoptado e passa a ser disseminado também pelo nó. Se nenhum escalonamento for encontrado, então o nó estabelece o seu próprio e começa a disseminá-lo através de pacotes SYNC.

Como os nós dormem a maior parte do tempo, é esperado que um grande número de nós tente aceder ao meio nos períodos ativos. O protocolo lida com esta situação fazendo com que os nós esperem um tempo aleatório antes da transmissão. É também usada a troca de pacotes RTS/CTS antes de cada transmissão para reduzir as colisões.

- ### T-MAC

O protocolo T-MAC (*Timeout Medium Access Control*) [12] tenta melhorar a rigidez do *duty-cycle* do protocolo S-MAC propondo um *duty-cycle* adaptativo, em que a duração dos períodos vai variando de acordo com o tráfego. A ideia principal do T-MAC é prever a atividade no meio para por o nó a dormir o mais cedo possível.

A grande vantagem da aplicação desta técnica é a redução considerável do consumo de energia. Tem também como desvantagem a possibilidade de um nó mudar para o estado de adormecido quando ainda há nós com mensagens para lhe enviar.

- ### MS-MAC

O protocolo MS-MAC (*Mobility aware* S-MAC) [13] propõe um mecanismo que adapta o *duty-cycle* do SMAC de modo a melhorar os tempos de arranque em redes móveis.

Os sensores medem as alterações de nível do sinal nos pacotes de SYNC que recebem e usam-nas para estimar a velocidade dos nós. Esta informação é depois disseminada através dos pacotes SYNC. Quando os nós vizinhos recebem os pacotes SYNC com indicação de maior velocidade, eles aumentam os períodos em que estão ativos. A mobilidade torna asligações transitórias, obrigando a reduzir o tempo a dormir para aumentar o débito durante o tempo em que a ligação está ativa. A mobilidade também provoca a necessidade frequente de descoberta e sincronização com novos vizinhos.

2.2.6. Protocolos com Preâmbulos

Os protocolos com preâmbulos não usam períodos comuns - em vez disso os sensores determinam o seu escalonamento independentemente. Nestes protocolos, os sensores passam a maior parte do tempo a dormir e acordam só o tempo suficiente para ver se existe alguma transmissão. Para não correrem o risco de perderem os dados, estes são precedidos de preâmbulos para avisar todos os receptores da transmissão. Para esta técnica funcionar, a duração dos preâmbulos tem que ser de pelo menos igual ao tempo entre duas verificações consecutivas do meio por parte dos sensores. Só assim é garantido que os sensores estão ativos durante o envio dos preâmbulos.

- B-MAC

O protocolo B-MAC (*Berkeley MAC*) [14] usa uma técnica de amostragem do meio e sua posterior comparação com um valor predefinido para ver se o canal se encontra ocupado ou não. Esta técnica depende de o sensor conseguir detectar corretamente o nível de sinal quando o canal não está a ser usado. O protocolo usa o controlo automático do ganho para se adaptar às mudanças no ambiente quando está a estimar o nível de ruído.

Além de evitar colisões e de ter uma boa utilização do meio esta técnica apresenta benefícios adicionais, tais como a possibilidade de um nó que está a escutar um preâmbulo enquanto espera pelos dados aperceber-se que o meio já não está a ser usado, evitar a recepção e ir dormir mais cedo.

- X-MAC

O protocolo X-MAC [15] tenta eliminar alguns dos problemas apresentados pelo protocolo B-MAC. Para baixar o consumo energético e tornar possível a realização em rádios baseados em pacotes (e.g. IEEE 802.15.4), os longos preâmbulos foram substituídos por uma sequência de preâmbulos de curta duração. Foi também adicionado ao preâmbulo o endereço de destino dos pacotes para reduzir o efeito de *overhearing*. Além destas alterações, este protocolo também apresenta um *duty-cycle* variável que pode ser adaptado ao tráfego existente na rede.

- WiseMAC

Preâmbulos de grandes dimensões diminuem o desempenho dos protocolos que os usam pois consomem muita energia. O protocolo WiseMAC (*Wireless Sensor* MAC) [16] reduz este problema fazendo com que seja possível usar preâmbulos de pequenas dimensões nas comunicações *unicast*. Este objectivo é atingido fazendo com que os sensores saibam a altura em que os seus vizinhos acordam e que enviem o preâmbulo ligeiramente antes, de modo a este ser o mais pequeno possível. Para manter este esquema simples, os sensores

enviam a informação de quando acordam nos pacotes de ACK usados para indicar a correta recepção dos dados.

2.2.7. Protocolos Híbridos

Estes protocolos combinam várias características dos protocolos descritos acima, utilizando as suas vantagens para atingir bom desempenho sobre condições de tráfego variadas. Quando o tráfego é baixo as técnicas de contenção têm desempenho suficiente mas quando o tráfego aumenta os protocolos de escalonamento apresentam melhor desempenho.

- #### Z-MAC

A solução do protocolo Z-MAC (*Zebra MAC*) [17] para as redes com tráfego variável é usar CSMA quando existe pouco tráfego e comutar para TDMA quando o tráfego aumenta.

Na fase inicial, os sensores correm um algoritmo de distribuição de *slots* para atribuir um *slot* a cada nó. Os *slots* atribuídos são suficientemente grandes para transmitir vários pacotes, fazendo com que seja possível haver deslizes no sincronismo. Os nós só transmitem nos seus *slots* assignados, mas quando necessitam tentam usar os *slots* não usados dos seus vizinhos. Isto é feito esperando um tempo aleatório, suficientemente grande para perceber que o *slot* não está a ser usado.

Visto o protocolo sofrer de deslizes no sincronismo, é necessário executar periodicamente o algoritmo de distribuição de *slots*, o que reduz a sua eficiência energética.

- #### MH-MAC

O protocolo MH-MAC (*Multimode Hybrid* - MAC) [18] é um protocolo híbrido apresentando três modos de funcionamento, modo FULL-ON, modo Assíncrono e modo Síncrono. O modo no qual o sensor funciona é controlado pela aplicação e por omissão este é o modo FULL-ON, no qual o sensor está permanentemente ativo.

No modo assíncrono é usado um algoritmo semelhante ao do X-MAC, sendo enviados preâmbulos de curta duração antes do envio de dados. Este melhora o desempenho apresentado pelo X-MAC introduzindo um campo adicional nos preâmbulos que contêm o tempo que falta para o envio de dados. Isto faz com que seja possível que os sensores durmam até ao instante exato da transmissão quando a transmissão é em *broadcast*, poupando assim energia. Para a redução de colisões entre pacotes de preâmbulos é usado um pacote Shut-Up. Este pacote é enviado caso um sensor receba preâmbulos vindos de diferentes emissores, e o seu objectivo é fazer com que apenas um emissor continue a transmitir.

No modo síncrono, o tempo é dividido em tramas e cada uma destas é dividida em onze *slots* de 100ms cada. Por omissão os sensores usam um *slot* público, onde todos os sensores podem comunicar, que é usado para a manutenção do sincronismo. Os restantes *slots* são privados e usados para comunicação *unicast* entre os sensores. Em todos os *slots* um protocolo semelhante ao observado em T-MAC é utilizado. Visto o número de *slots* ser fixo, o sincronismo entre os vários sensores é mantido com a distribuição de pacotes SYNC que indicam o início do *duty-cycle* do sensor.

É possível a utilização de sensores em diferentes modos de funcionamento na vizinhança uns dos outros, bastando para isso que os sensores em modo não síncrono mantenham uma tabela com o escalonamento dos *slots* públicos dos sensores síncronos, e apenas usem esses *slots* para a transmissão de dados.

- ### Funneling-MAC

Nas redes de recolha de dados, as redes podem apresentar um efeito de funil visto que todo o tráfego segue na direção de um nó central. O protocolo Funnelling-MAC [19] usa TDMA na região perto do receptor central e CSMA no resto da rede. Como todo o tráfego converge para o receptor central, o tráfego à sua volta é muito intenso e cerca de 80% da perda de pacotes acontece ai [20]. Fora dessa zona, o tráfego é menos intenso e CSMA é usado para obter um melhor desempenho.

Capítulo 3. Sistema Proposto

Neste capítulo é apresentada a arquitetura do protocolo proposto e é descrita a abordagem usada para realizar o protocolo na plataforma de sensores sem fios SunSPOT.

São descritos os componentes que suportaram a implementação, para além da implementação propriamente dita. Também são realçados os problemas encontrados e as opções tomadas para os solucionar ou mitigar.

3.1. Descrição formal do protocolo realizado

O protocolo MMH-MAC é um protocolo complexo com dois modos de funcionamento, um assíncrono e outro síncrono. Visto o protocolo não ter sido implementado na sua totalidade, apenas é seguidamente descrita a parte realizada.

O modo assíncrono executa um algoritmo LPL (*Low Power Listening*) semelhante ao existente no X-MAC. É enviada uma sequência de pequenos preâmbulos durante um período que pode ser no máximo igual a duas vezes o ciclo de funcionamento, que neste caso é de 1100ms, antes de enviar um pacote de dados. O envio dos preâmbulos assegura que o receptor está acordado quando o pacote de dados é enviado.

Os preâmbulos contêm o endereço de destino e são separados por pausas de modo a ser possível aos receptores *unicast* enviarem um pacote PACK (*Preamble* ACK) indicando que está pronto a receber os dados. Isto faz com que a duração da sequência de preâmbulos seja reduzida.

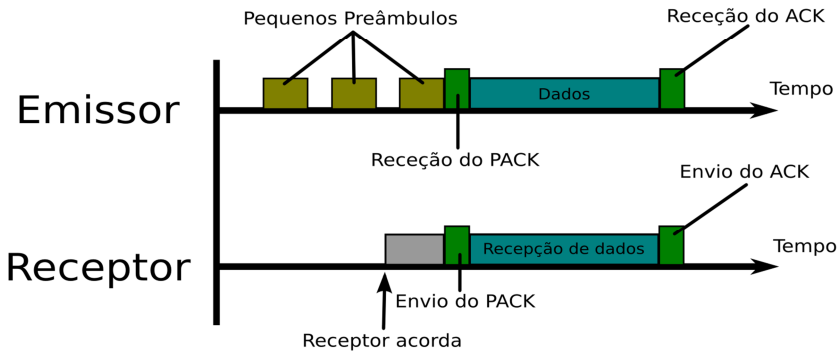

Figura 3.1 – Transmissão de dados assíncrona em *unicast*

Em casos de *broadcast* é necessário o envio de todos os preâmbulos da sequência antes do envio do pacote, para garantir que todos os receptores do pacote estão activos durante a sua transmissão.

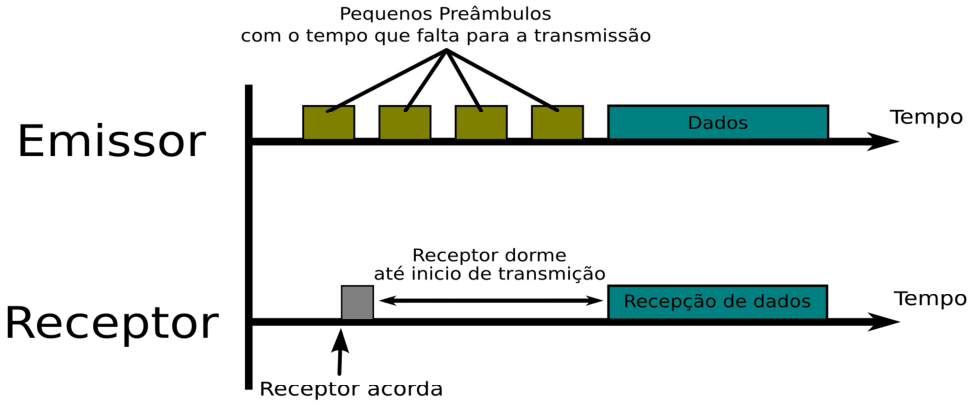

Figura 3.2 – Transmissão assíncrona de dados em *broadcast*

O protocolo também melhora a proteção de *overhearing* existente em X-MAC, adicionando um campo no preâmbulo que indica o tempo que falta para o início da transmissão dos dados. Em caso de *broadcast* os receptores usam este valor para dormir até ao envio dos dados. Outros sensores que estejam à espera para enviar dados e recebam preâmbulos destinados a outro sensor também usam este valor para controlar o tempo a dormir, reduzindo o *overhearing*.

O protocolo lida com a colisão entre preâmbulos usando pacotes do tipo SHUT-UP. Os receptores enviam um pacote deste tipo com uma probabilidade *p* quando recebem preâmbulos de mais do que um emissor. Este pacote inclui o endereço do emissor ativo e o tempo que falta para terminar o envio dos preâmbulos, permitindo aos sensores que o recebem abortar a emissão dos preâmbulos e dormir até a transmissão terminar.

O tempo entre preâmbulos inclui um *jitter* para diminuir probabilidade de existirem colisões entre os preâmbulos quando mais de dois nós transmitem preâmbulos em simultâneo.

A recepção de pacotes é apenas confirmada quando estes são transmitidos em *unicast,* usando pacotes ACK. Depois da recepção dos dados, o receptor mantem-se acordado durante algum tempo, à espera de novos pacotes, antes de voltar a dormir.

3.2. AnyMAC

A plataforma SunSPOT apresenta a limitação de não disponibilizar mecanismos simples para o desenvolvimento de procotolos MAC. Este desenvolvimento geralmente tem de ser feito ao nível do código da máquina virtual. Nesta dissertação foi usada uma abordagem alternativa, de desenvolver o protocolo sobre a plataforma AnyMAC [21]

A plataforma AnyMAC foi desenvolvida no contexto de uma tese de mestrado, colmatando um défice no desenvolvimento de protocolos MAC na plataforma de sensores sem fios SunSPOT. Este défice reside na implementação dos protocolos de comunicação que vêm por omissão nos sensores SunSPOT, que não providenciam uma separação clara entre eles - os vários protocolos de comunicação estão todos interligados, especialmente nos vários níveis de rede.

O AnyMAC providencia uma clara separação dos níveis físico, lógico e rede e uma interface de programação para aceder a cada um destes níveis, fazendo com que a implementação de protocolo de acesso ao meio possa ser facilmente trocada.

3.2.1. Características do AnyMAC

Seguidamente são descritas as principais características escolhidas a quando do desenho da implementação do AnyMAC.

- Permutabilidade

  Deve ser possível trocar o protocolo de acesso ao meio facilmente sem que isso tenha impacto no resto da implementação. A única ressalva é quando são usadas especificidades do protocolo implementado fora do nível do protocolo.

- Flexível para implementação de novos protocolos

  A implementação deve ser flexível o suficiente para permitir a fácil implementação de novos protocolos para a plataforma. Um exemplo é a possibilidade de usar um formato de pacotes diferente do definido por omissão.

- Transparência para a aplicação

  As aplicações que foram desenvolvidas para funcionar com a implementação padrão da plataforma devem continuar a funcionar com mínimo de alterações possíveis quando a plataforma é estendida pelo AnyMAC.

- Permitir que as aplicações usem especificidades do protocolo

Apesar de tentar ser transparente para as aplicações deve ser também possível usar especificidades do protocolo, se assim for desejado.

- Compatibilidade com a implementação original

Apesar de todas as alterações a implementação deve ser sempre que possível compatível com a implementação original. Este objectivo é atingido fazendo o mínimo de modificações possíveis no resto da implementação, que não envolve o protocolo de acesso ao meio.

3.2.2. Arquitetura dos SunSPOT

Na Figura 3.3 apresentada abaixo é possível observar a arquitetura original da plataforma dos SunSPOT.

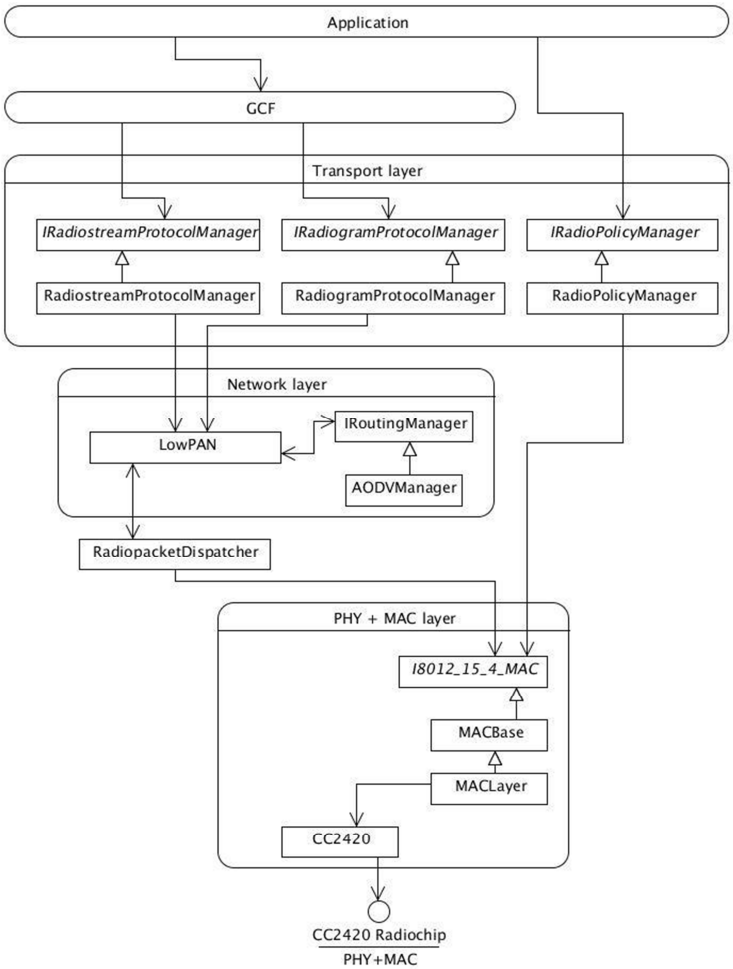

Figura 3.3 – Arquitetura dos sensores sem fio SunSPOT

Esta figura apresenta os principais componentes dos níveis de rede, os quais são descritos seguidamente.

- O integrado de rádio CC2420 é um rádio transcetor (*transceiver*) compatível com o protocolo IEEE 802.15.4 que é usado pela plataforma para comunicação rádio. Ele providencia funcionalidade da camada física e parcialmente do protocolo MAC da especificação IEEE 802.15.4.

- A classe *CC2420* exporta a funcionalidade do CC2420 para o resto dos componentes. Esta classe é responsável pela transmissão e recepção dos pacotes. Esta classe também permite que vários parâmetros, tais como o canal e a potência do sinal de saída, sejam configuradas no rádio.

- As classes *MACBase* e *MACLayer* juntas implementam toda a funcionalidade MAC que não é providenciada pelo integrado CC2420. Estas duas classes conjuntamente com a classe *CC2420* formam a implementação da especificação IEEE 802.15.4 na plataforma.

- A *RadioPacketDispatcher* é uma classe intermédia entre o nível MAC e o nível de rede. Ela é responsável por transferir os pacotes entre estes dois níveis.

- A classe *LowPAN* providencia uma implementação parcial de 6LowPAN, a qual é usada como nível de rede na plataforma. Ela providencia, entre outras coisas, fragmentação de pacotes e difusão multi-salto (*multihop broadcasting*). A classe também permite que os vários gestores de protocolos (*ProtocolManagers*) se registem para um tipo específico, sendo posteriormente os pacotes recebidos distribuídos para o *ProtocolHandler* correspondente. Este mecanismo é baseado no princípio "*multiple header*" definido na especificação LowPAN.

- O *AODVManager* (que foi removido nas versões mais recentes) implementa o protocolo de encaminhamento AODV usado pela plataforma.

- As classes *RadiostreamprotocolManager* e *RadiogramProtocolManager* são registadas como *ProtocolManagers* na classe *LowPAN*. Estas classes implementam os protocolos de *radiostream* (em feixe) e de *radiogram* (com datagramas) do nível de transporte. Estas classes não são acedidas diretamente pela aplicação; uma camada de abstração chamada de GCF está localizada entre o nível de transporte e o nível de aplicação. Esta camada é responsável por disponibilizar diferentes recursos, tais como o rádio e a USB, de uma forma universal.

- A classe RadioPolicyManager é responsável pelo controlo do rádio, sendo ela que decide quando é que o rádio é ligado ou desligado. Apesar de esta classe não implementar o nível de transporte, ela é usada indiretamente pela classe Radiostream no RadiogramProtocolManager. Como resultado, ela é considerada uma parte integral do nível de transporte.

- ## Deficiências

O desenho dos níveis de rede da plataforma SunSPOT tem algumas deficiências relacionadas com o desenvolvimento de protocolos de controlo de acesso ao meio. A maioria dessas deficiências é o resultado direto da plataforma SunSPOT ter sido optimizada para ser usada com o protocolo de controlo de acesso ao meio IEEE 802.15.4. O resultado dessas deficiências manifesta-se de várias maneiras:

- A classe *CC2420* foi desenhada para ser usada com o protocolo IEEE 802.15.4 e por consequência todos os pacotes que não aderem ao formato desse protocolo são automaticamente descartados pelo rádio.

- Uma única classe (a classe *RadioPacket*) é usada para representar os pacotes no nível físico, de dados e na classe *LowPAN* descrita acima. Além disso, a estrutura interna que guarda os dados do pacote é acedida diretamente por componentes diferentes nestes três sítios. Isto faz com que qualquer mudança à estrutura do pacote implique mudanças em muitos locais diferentes o que se torna propício a erros.

- O formato dos endereços do protocolo IEEE 802.15.4 é usado em todos os níveis da plataforma. Consequentemente qualquer tentativa de desenvolver um protocolo de controlo de acesso ao meio tem que manter este formato.

3.2.3. Arquitetura do AnyMAC

Na Figura 3.4 apresentada abaixo é possível observar a arquitetura do AnyMAC.

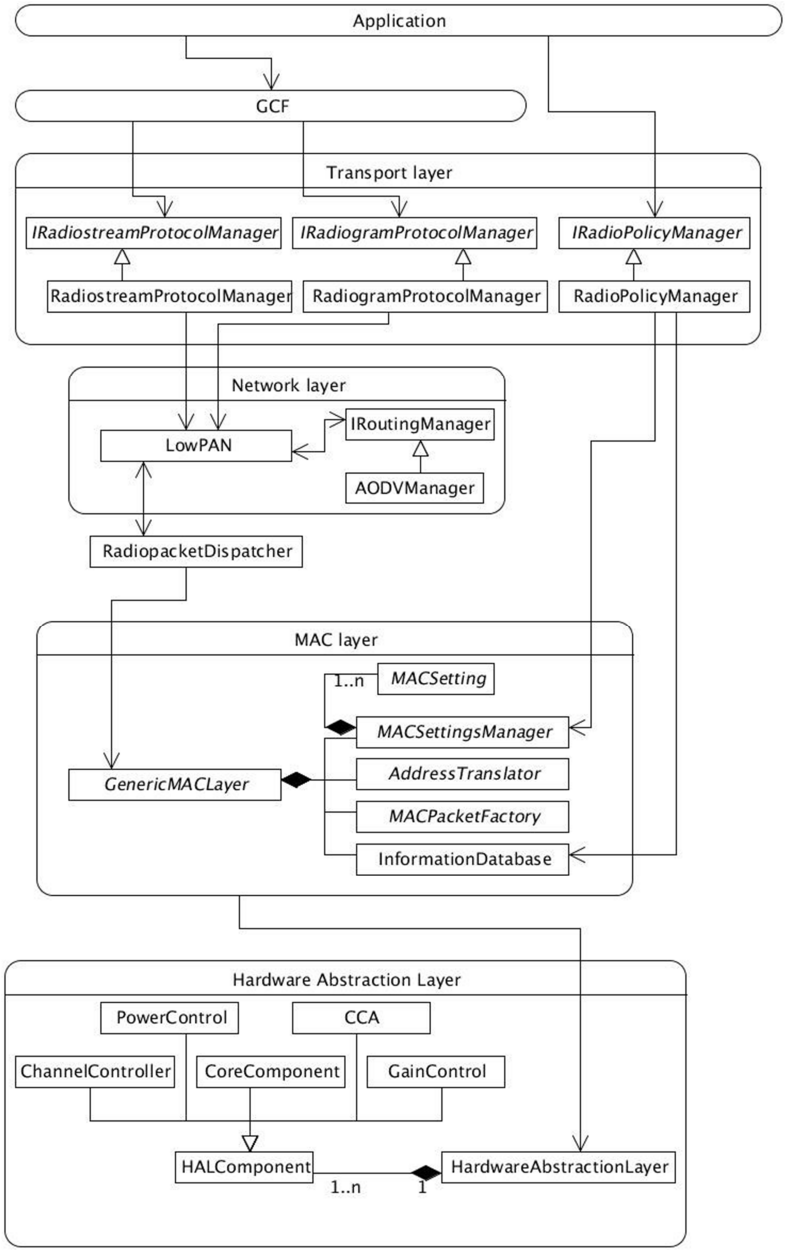

Figura 3.4 – Arquitetura do AnyMAC

Como é possível observar pela figura acima, as camadas superiores do AnyMAC são idênticas às originalmente na plataforma. A camada física e de MAC forma substituídas pela *Hardware Abstraction Layer* e uma nova camada MAC. A forma como os pacotes são manipulados também foi drasticamente modificada.

- Formato dos pacotes

De modo a criar uma clara separação entre camadas, um pacote não é representado por uma única classe, mas sim por uma classe diferente em cada nível da rede. Apesar da representação interna continuar a ser a mesma, o acesso à mesma só é possível através da classe *PacketPayload*. Esta classe garante que uma classe que represente o pacote num determinado nível da rede não pode aceder à representação interna do pacote que pertence a níveis inferiores.

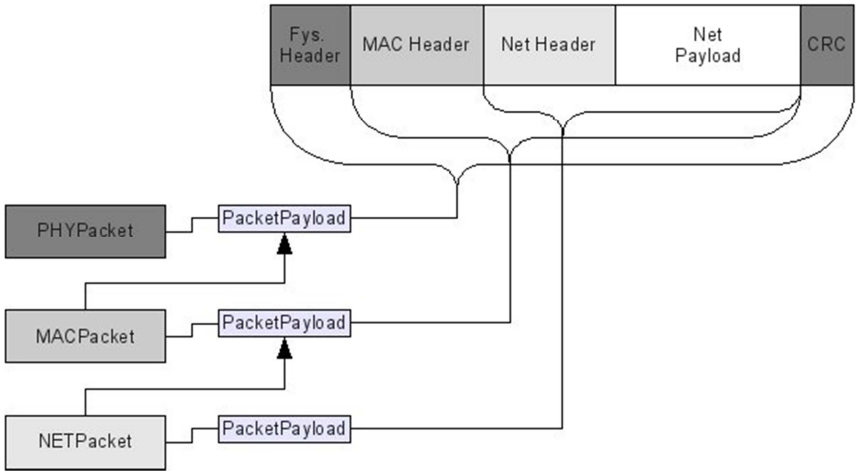

Figura 3.5 – Formato dos Pacotes

Por outros termos, isto significa que cada nível só pode aceder à região de *payload* (dados) do nível abaixo. Este mecanismo é ilustrado na figura seguinte.

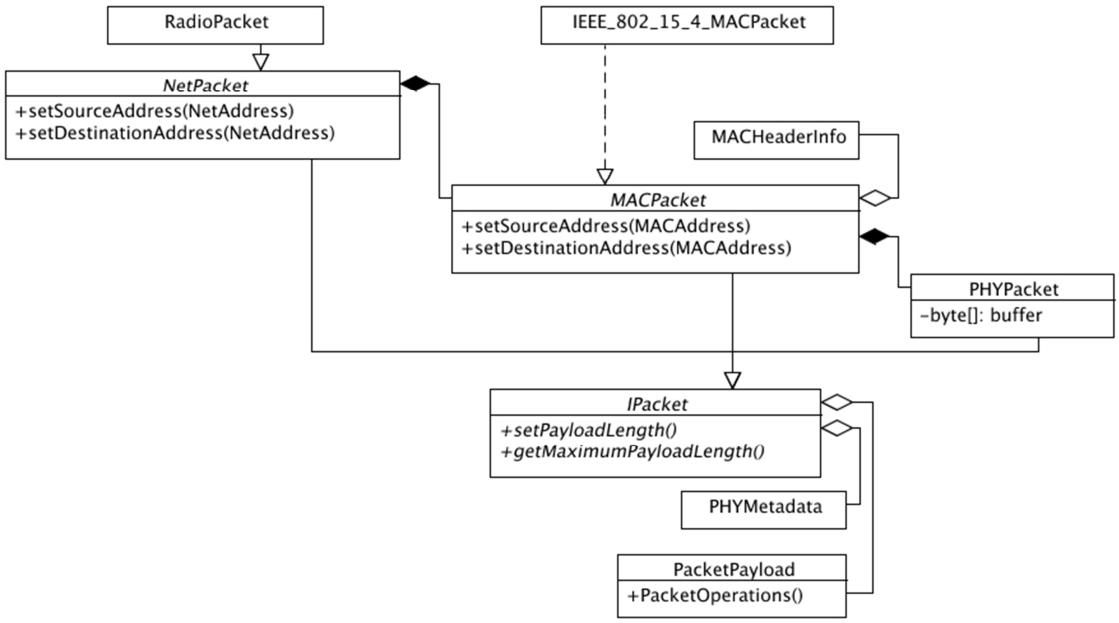

Figura 3.6 – Arquitectura dos Pacotes

Seguidamente apresentam-se as classes mais importantes desta arquitetura:

- A classe *PacketPayload* é responsável por controlar a representação interna dos pacotes. Ela tem todos os métodos necessários para trabalhar com a representação interna.

- A interface *IPacket* deve ser implementada por todas as classes que querem representar um pacote num nível particular. Ela tem métodos para obter o *PacketPayload* de um pacote e para obter e atribuir o comprimento do *payload*.

- A classe *PHYPacket* representa um pacote no nível físico e é a classe que realmente contém a representação interna do pacote. O acesso a esta representação é controlado pela classe *PacketPayload* associada ao pacote.

- A classe abstracta *MACPacket* representa um pacote no nível de dados. Esta serve de base para o desenvolvimento de novos formatos de pacotes. Todas as classes que derivem desta devem implementar a interface *IPacket* e os métodos necessários para obter e atribuir os endereços do emissor e do receptor.

- A classe *NetPacket* serve como base para a definição de novos pacotes no nível de rede. Tal como com a classe *MACPacket*, classes que derivem desta devem implementar a interface *IPacket*. Esta classe também define métodos abstractos para obter e atribuir os endereços do receptor e do emissor. De modo a providenciar compatibilidade com a implementação *LowPAN* existente, a classe *RadioPacket* existente na implementação original da plataforma é agora uma extensão a esta classe.

- ### *Hardware Abstraction Layer*

Em contraste com a classe CC2420 da implementação original, a *Hardware Abstraction Layer* (HAL), não foi desenvolvida com as necessidades de um protocolo MAC em mente. Em vez disso, ela expõe a maioria das possibilidades do rádio CC2420 para as camadas superiores, aumentando assim a flexibilidade.

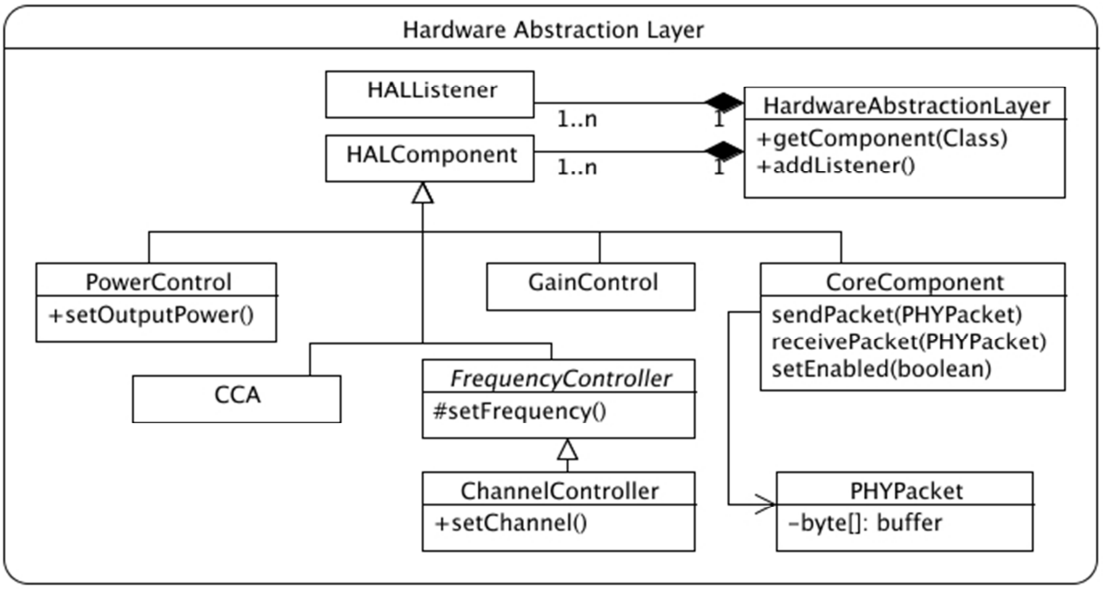

Figura 3.7 – Arquitetura da *Hardware Abstraction Layer*

Como é possível observar pela figura acima, o HAL é composto por vários componentes, sendo cada um deles responsável por uma única funcionalidade do rádio, tal como o envio de pacotes, mudar o canal e configurar a potência de envio. Nem todos os componentes têm que ser inicializados no arranque, fazendo com que seja possível para quem está a implementar um novo protocolo MAC decidir quais os componentes que vai usar, dependendo dos seus requisitos. É ainda possível adicionar novos componentes ou substituir os existentes.

Os seguintes componentes do HAL (*HALComponents*) são providenciados por omissão:

- O *CoreComponent* é o componente que contem a funcionalidade fundamental do HAL. Ele é responsável por enviar e receber *PHYPackets* e de ligar e desligar o rádio CC2420. Em contraste com todos os outros *HALComponents*, este componente está sempre presente no HAL e não pode ser substituído. A sua implementação é genérica e extensível o suficiente para poder suportar quase todas as implementações MAC.

- O *FrequencyController* é uma classe abstracta que permite a mudança de frequência a que o rádio opera seja modificada. De acordo com a especificação o CC2420 não só é capaz de funcionar às frequências definidas na especificação IEEE 802.15.4 mas também a muitas outras frequências. Ao estender esta classe é possível definir as frequências usadas na comunicação.

- O *ChannelController* é derivado do *FrequencyController HALComponent* e providencia canais que estão definidos na especificação IEEE 802.15.4.

- A classe *ClearChannelAssessment* permite ao utilizador modificar vários parâmetros do algoritmo CCA usado pelo rádio. Também permite ao utilizador verificar se o meio está livre.

- A classe *GainControl* permite ao utilizador mudar as definições de controlo de ganho do rádio.

- O *PowerController* permite ao utilizador determinar a energia gasta no envio de pacotes.

- MAC

De modo a ser possível a fácil substituição do protocolo MAC usado, a arquitetura do AnyMAC providencia uma série de classes interfaces e classes abstractas nomeadas de 'MAC layer API' (Interface de Programação de Aplicações da camada MAC). É apenas através desta API que a implementação pode ser acedida.

A Figura 3.8 mostra a arquitetura desta API juntamente com uma possível implementação.

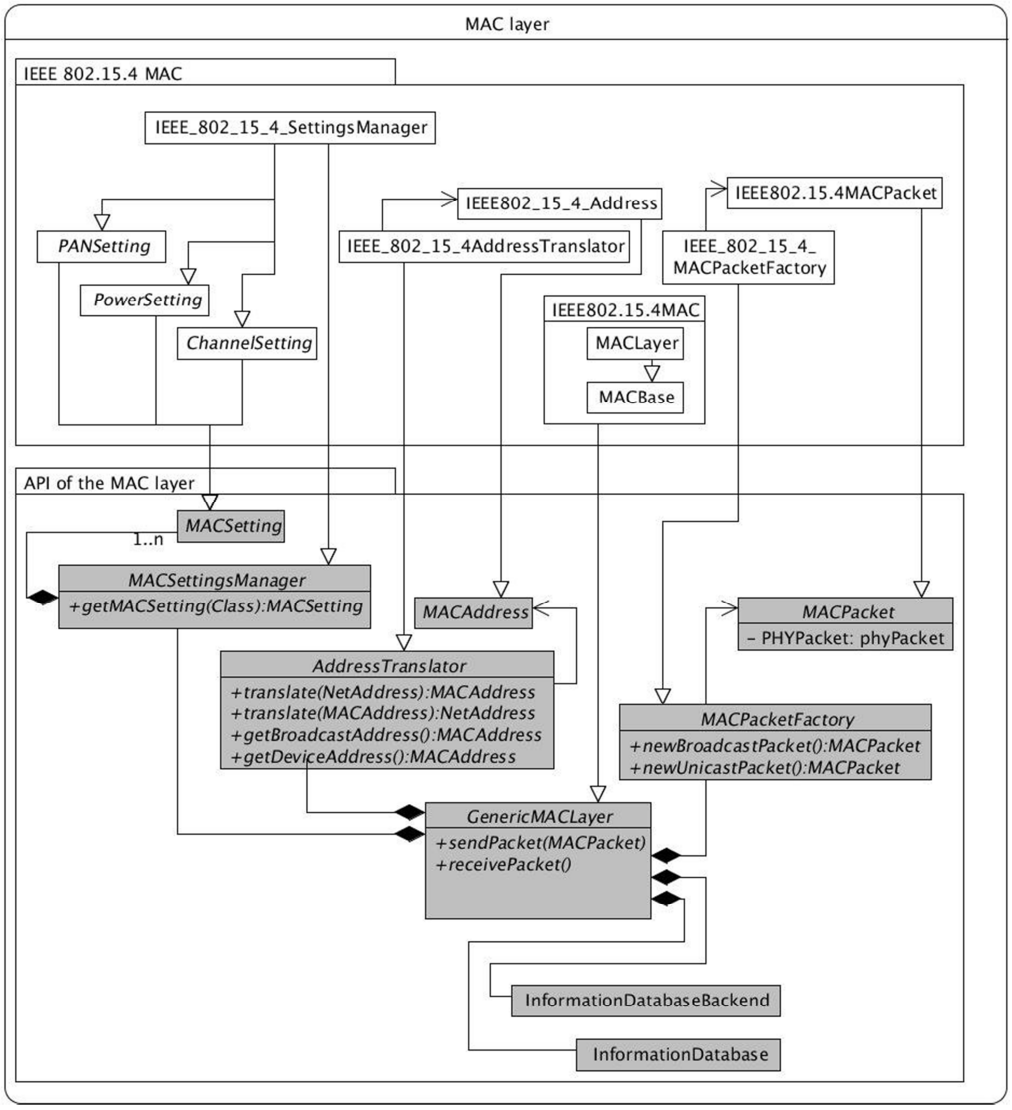

Figura 3.8 – Arquitetura do MAC

Como é possível ver pela figura, a API define múltiplos componentes que devem ser providenciados por cada implementação MAC. Os componentes são:

- A classe abstracta *GenericMACLayer* é o componente central de cada possível implementação. Ela tem os métodos necessários para providenciar acesso aos

outros componentes da implementação, mas também atua como uma fachada de modo a que a funcionalidade providenciada pelos outros possa ser usada diretamente. Este componente também providencia a funcionalidade central de cada implementação, enviando e recebendo os pacotes e ligando e desligando o rádio. A restante funcionalidade é providenciada por outros componentes.

- Um objecto *AddressTranslator* é adicionado à API de modo a ser possível definir o formato dos endereços. Na implementação original os endereços de 64 bits da especificação IEEE 802.15.4 eram usados em todos os níveis da rede para identificar o sensor na rede. Como o protocolo LowPAN, que opera acima do nível de dados, espera que o protocolo MAC use o formato de endereço da especificação IEEE 802.15.4, um mecanismo de tradução é usado para traduzir entre o formato usado no protocolo MAC e o esperado pelo LowPAN. Este mecanismo é acedido através do *AddressTranslator*. É assim possível definir um formato de endereços próprio com a única condição de ter que providenciar um mecanismo de tradução. Este mecanismo pode ser implementado usando por exemplo *hashing*, tabelas ou mesmo uma variação do protocolo ARP.

- A classe *MACPacketFactory* é uma fábrica abstracta para a criação de pacotes. Ela é usada para permitir que os níveis superiores da rede sejam capazes de criar pacotes (*MACPackets*) sem ser necessário ter o conhecimento do formato dos mesmos.

- O *MACSettingsManager* é usado para permitir que as aplicações usem especificidades dos protocolos MAC sem ser necessário que os níveis superiores estejam cientes de qual implementação é usada. Cada implementação MAC pode definir zero ou mais interfaces *MACSetting*. Podem ser também criadas novas interfaces para descrever funcionalidade que não seja descrita pelas interfaces existentes. As aplicações podem então pedir ao *MACSettingsManager* (o qual está diretamente disponível através do *RadioPolicyManager*) que indique se a implementação MAC implementa ou não uma certa interface. Por exemplo, a implementação MAC IEEE 802.15.4 implementa as interfaces *ChannelSetting* e *PANSetting*. Quando uma implementação MAC que não suporta o uso de identificadores PAN é usada, a interface *PANSetting* não está disponível.

- InformationDatabase

Este é um componente da API que foi desenhado para facilitar a passagem de informação através dos níveis. Este componente é disponibilizado para o utilizador e para a implementação do protocolo MAC, e é portanto o único componente que não necessita de ser providenciado pela implementação MAC. Funciona com uma base de dados em memória onde a informação é guardada.

Como é visível na Figura 3.9, este componente é acedido através de duas interfaces separadas. A interface *InformationDatabase* é providenciada ao utilizador para que seja possível guardar e obter informação baseada num identificador de texto (*String*). Os utilizadores também podem registar um *InformationChangeListener* de modo a serem notificados sempre que uma informação específica muda. A interface *InformationDatabaseBackend*, por outro lado, é usada pela implementação MAC e permite que a informação seja guardada e atualizada.

Por razões de eficiência o *InformationDatabaseBackend* indexa a informação usando um número único (*integer*) em vez de usar texto.

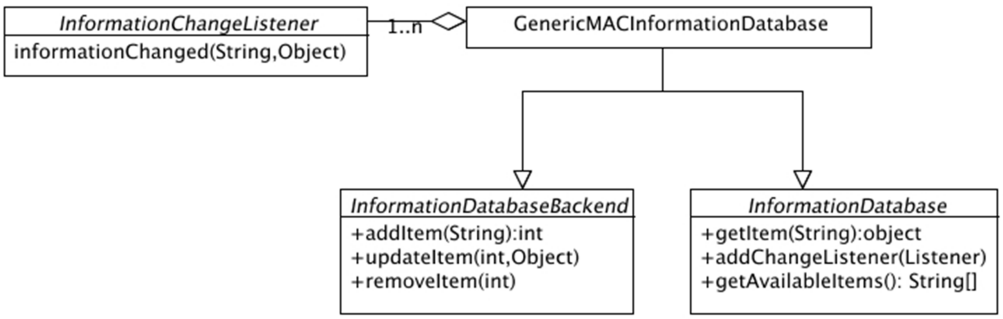

Figura 3.9 – InformationDatabase

3.3. Implementação do MH-MAC em AnyMAC

A implementação parcial do protocolo MMH-MAC foi realizada com base num exemplo de um protocolo de acesso ao meio existente em AnyMAC, o DemoMAC. Este exemplo foi duplicado e depois iterativamente modificado até ter sido completamente substituído pelo protocolo a implementar.

Começou-se por implementar extensões às classes bases já descritas acima, *GenericMACLayer*, *AddressTranslator*, *MACPacketFactory* e *MACSettingsManager*.

Tal como descrito acima, a classe *GenericMACLayer* é o componente central da implementação. Esta classe foi estendida sendo criada a classe MHMAC. A principal funcionalidade desta classe é a de providenciar os pontos de entrada onde podem ser introduzidas as alterações ao protocolo de acesso ao meio. Os pontos de entrada são:

- O método *initialize* é usado para inicializar o protocolo usando os componentes HAL que recebe como parâmetro. É aqui que também são inicializadas todas as estruturas e classes auxiliares. São aqui também inicializadas tarefas (*threads*) de recepção e envio de pacotes.

- O método *requiredHALComponents* apenas retorna uma lista de classes, cada uma delas representando um componente que se quer usar. Nesta implementação apenas se usou o componente para controlo de potência de sinal e o componente para escolha do canal através das classes existentes no AnyMAC, a classe *PowerController* e a classe *ChannelController*.

- O método *getPacketFactory* devolve uma classe capaz de criar pacotes do protocolo MAC, e é usada pelos protocolos dos níveis superiores para a criação destes. Os pacotes criados por esta classe são os novos pacotes *MHMACPacket* que foram definidos nesta implementação.

- O método *getTranslator* retorna uma implementação do *AddressTranslator* descrito acima. A implementação retornada neste caso é a classe *MHMACAddressTranslator* que faz a conversão entre os endereços usados no nível MAC e os usados no nível de rede. Esta classe também é usada quando outros componentes necessitam de saber qual é o endereço de *broadcast* ou o endereço do próprio sensor.

- O método *getSettingsManager* devolve uma implementação de *MACSettingsManager* descrito acima. Nesta implementação a classe devolvida é a classe *MHMACSettingsManager.*

- O método *sendPacket* é chamado pela aplicação para o envio de um pacote. A chamada deste método desencadeia todo o processo de envio de pacotes especificado em 3.1.

- O método *receivePacket* também é chamado pela aplicação para a recepção de um pacote. Este método bloqueia enquanto não houver pacotes.

- O método *isReceiverEnabled* indica se a camada MAC é capaz de receber pacotes.

- O método *setReceiverEnabled* é usado para ativar ou desativar a recepção de pacotes na camada MAC.

Também foi implementada uma extensão à classe *MACSettingsManager* - a classe *MHMACSettingsManager*. Como descrito anteriormente, esta extensão inicializa os componentes necessários para o HAL, que no caso desta implementação foram os componentes de controlo de potência e de controlo de canal representados pelas classes *PowerController* e *ChannelController*. Estes componentes foram inicializados usando classes já existentes no AnyMAC: a classe *SimpleChannelSetting* e a classe *SimplePowerSetting*.

A extensão à classe *MACPacketFactory* tomou forma na classe *MHMACPacketFactory*. A única responsabilidade desta classe é a da criação de pacotes de dados. Para tal, ela é composta por apenas dois métodos, um para a criação de pacotes usados em *broadcast* e outro para criação de pacotes usados em *unicast*. Na criação de pacotes para *unicast* são definidos os valores de dois campos: o do endereço de destino, que recebe o valor do endereço de *unicast*, e o do tipo de pacote, que recebe o valor que indica que é um pacote de dados. Na criação de pacotes para *broadcast* é apenas definido o tipo de pacote como pacote de dados.

A classe *MHMACAddressTranslator* é a extensão ao *AddressTranslator*. Esta classe faz a conversão entre os endereços de nível MAC e os endereços de nível de rede. Nesta implementação não foram usados os endereços definidos em IEEE 802.15.4 e foi criado um novo formato. Este novo formato foi criado de modo a simplificar a implementação e o formato escolhido consiste num objecto do tipo *long* de Java. A classe *MHMACAddress* integra este novo formato em AnyMAC.

Devido às necessidades do protocolo implementado, também foi definido um novo tipo de pacote. O pacote pode ser de um de quatro tipos:

- O tipo DATA_TYPE é usado quando o pacote representa um pacote de dados;

- O tipo ACK_TYPE é usado quando o pacote representa um pacote ACK, usado para indicar a correta recepção de um pacote de dados;

- O tipo PREAMBLE_TYPE é usado quando o pacote representa um pacote de preâmbulo;

- O tipo PREAMBLE_ACK_TYPE é usado quando o pacote representa um pacote de ACK para preâmbulos, usado para terminar o envio dos preâmbulos antecipadamente.

Além do campo que indica o tipo de pacote também existe: um campo para o endereço de destino, um campo para o endereço do emissor, um campo que indica o número sequência do pacote e um campo com o número de sequência de grupo. O número de sequência é representado internamente por um byte e é incrementado a cada envio de um pacote do tipo DATA_TYPE. O número de sequência de grupo é usado para agrupar os preâmbulos correspondentes a um pacote. Todos os preâmbulos para um pacote de dados contêm o mesmo número de sequência de grupo. Em todos os pacotes só os campos relevantes para o tipo de pacote é que são preenchidos e o espaço restante no pacote pode ser usado para dados.

Além das classes descritas acima, foi também criado um conjunto de classes especificamente para lidar com o envio e recepção de pacotes. As classes *MHMACPacketReceiver* e *MHMACPacketSender* são usadas para a recepção e envio de pacotes respectivamente. A classe *MHMACPacketReceiver* é executada numa *thread* independente e o seu ciclo de execução apenas tem a tarefa de esperar pela recepção de pacotes. Aquando da chegada de pacotes, o objeto da classe notifica o ouvinte registado associado ao tipo de pacote. Existem quatro tipos de ouvintes interessados na recepção de pacotes, associados aos quatro tipos de pacotes existentes. A classe *MHMACPacketSender* também é executada numa *thread* independente e o seu ciclo de execução tem a tarefa de ver se existem pacotes para enviar e em caso positivo proceder ao envio dos mesmos. Também existem ouvintes registados no envio de pacotes e estes são dois, um para os pacotes do tipo ACK_TYPE e PREAMBLE_ACK_TYPE e outro para os restantes.

Tal como já mencionado acima, o uso de AnyMAC teve bastantes vantagens mas também apresentou algumas dificuldades. Uma das grandes dificuldades encontradas foi o sincronismo entre as várias *threads* de execução existentes. Os exemplos apresentados por AnyMAC não apresentavam estes problemas, pois em nenhum deles era necessário interromper a emissão de pacotes aquando da recepção de um tipo específico de pacotes como acontece com a recepção de ACKs para preâmbulos. Apesar de o código do AnyMAC estar disponível e existir uma breve descrição na internet [21], a tese de mestrado associada está escrita numa língua não falada pelo aluno que realizou a dissertação, tornando inviável a sua leitura. Assim, a maior parte da aprendizagem do AnyMAC fez-se através da análise do código.

Outro problema de grande importância foi que o ambiente de simulação providenciado pela plataforma SunSPOT não suporta o pacote AnyMAC. Ainda foi ponderada a implementação do

AnyMAC no ambiente de simulação, mas esta hipótese foi rejeitada devido à sua complexidade, esta saía muito do âmbito deste trabalho, e à falta de disponibilidade para realizar esta tarefa.

Capítulo 4. Análise de Desempenho

4.1. Ambiente de Teste

Como não foi possível o uso do ambiente de simulação, foram implementadas várias aplicações simples com o intuito de ajudar a desenvolver e depurar o protocolo durante a sua implementação. Estas aplicações foram também usadas para a análise de desempenho do protocolo.

O cenário usado na recolha de valores de desempenho foi composto por três sensores: um receptor central ligado ao PC e dois sensores independentes. O sensor que se encontrava ligado ao PC apenas abria uma ligação no porto escolhido e esperava passivamente pela recepção de pacotes enviados pelos outros dois sensores.

Os outros dois sensores tinham unicamente a função de enviar periodicamente os seus dados.

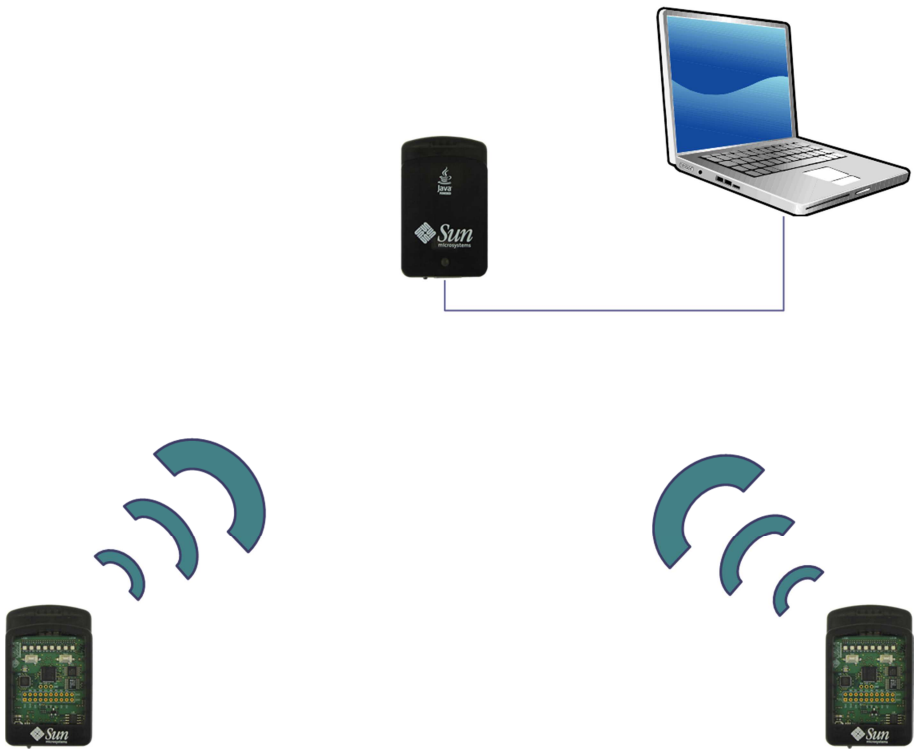

Figura 4.1 – Ambiente de teste

4.2. Medições Realizadas

Visto a plataforma dos SunSPOT ser bastante completa e fornecer acesso aos vários elementos do sensor, foi possível recolher vários tipos de medições. Foram medidos o nível da bateria dos sensores, a temperatura dos sensores e o tempo passado pelo sensor nos vários estados possíveis, ativo, a dormir desligando apenas parte dos componentes e a dormir desligando todos os componentes.

Estes valores foram registados executando as aplicações e deixando-as a funcionar durante sensivelmente uma hora. Na secção seguinte são apresentados os resultados obtidos.

4.3. Resultados

Seguidamente são apresentadas as medições realizadas com o protótipo realizado.

4.3.1. Temperatura

O gráfico seguinte representa o nível de temperatura do sensor ao longo do tempo.

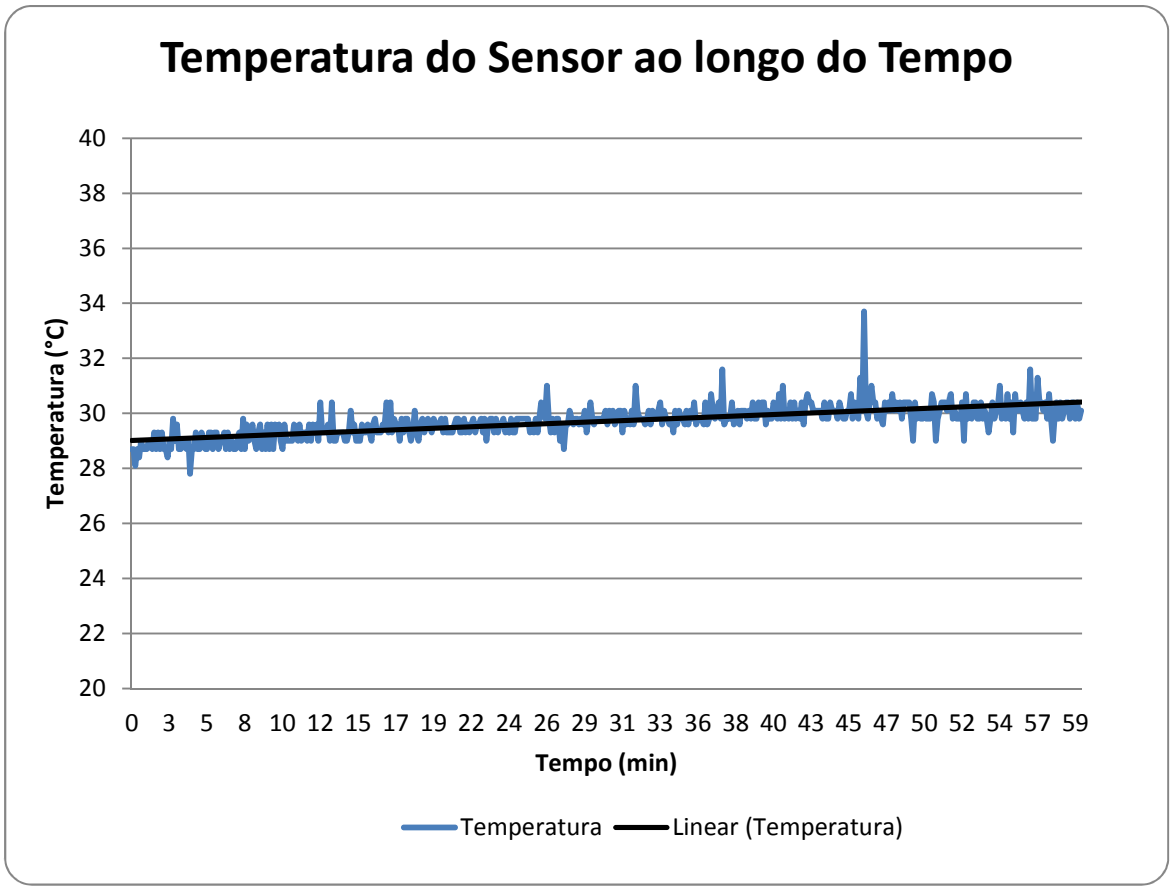

Figura 4.2 – Gráfico da Temperatura medida ao longo do Tempo

Como se pode observar a temperatura do sensor mantém-se sensivelmente constante ao longo do tempo.

4.3.2. Estado do sensor sem fios

Os sensores SunSPOT realizam dois estados de poupança de energia: sono superficial (*shallow sleep*) e sono profundo (*deep sleep*). O segundo modo desliga mais componentes do sensor sem fios, permitindo uma maior poupança energética. O gráfico seguinte representa a percentagem de tempo que o sensor passa nos vários estados.

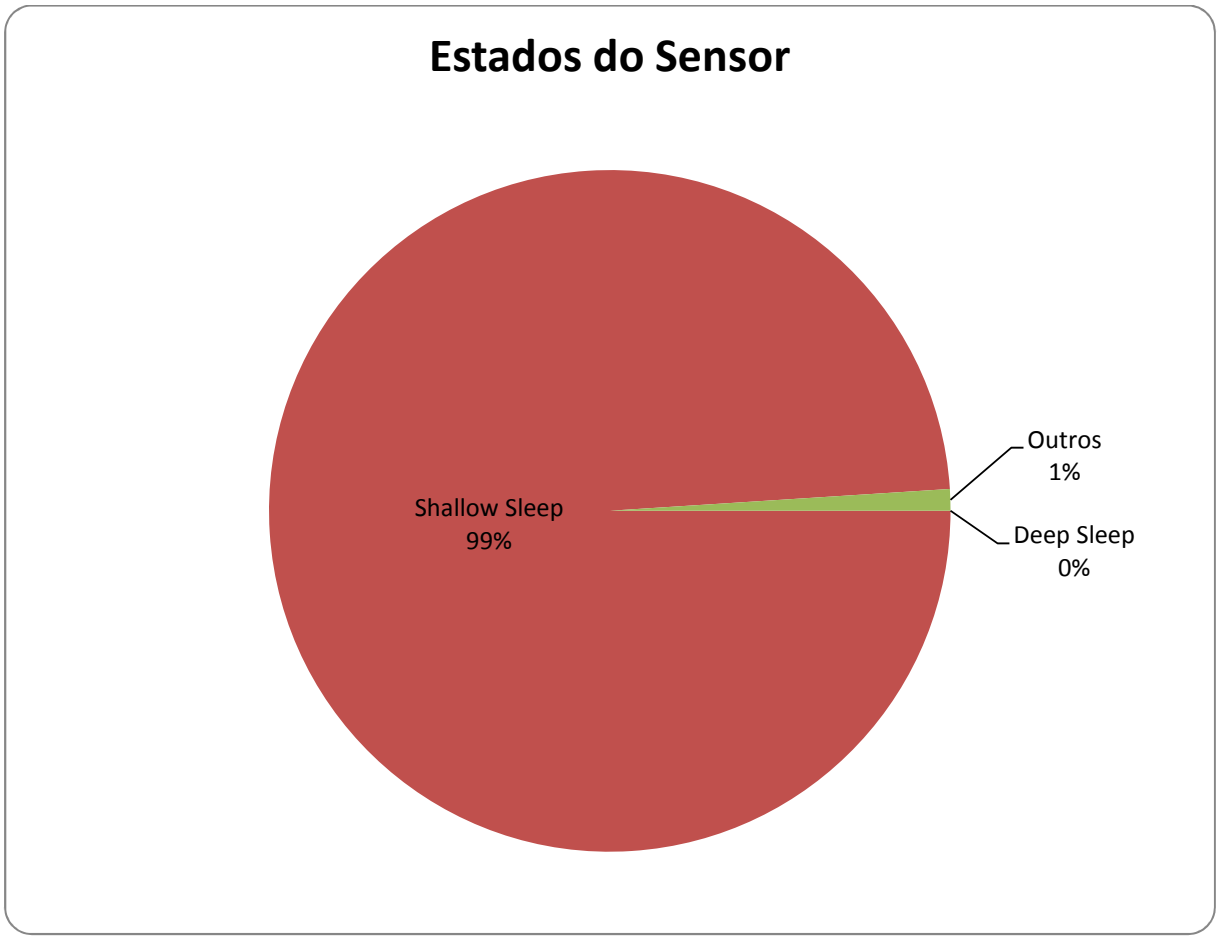

Figura 4.3 – Gráfico dos Estados do Sensor

Como se pode observar pelo gráfico, o sensor passa quase todo o seu tempo no primeiro estado de poupança de energia. Infelizmente não foi possível colocar o sensor no estado de sono profundo, em que este poupa mais energia. Este assunto foi investigado mas nunca resolvido e o pressuposto do autor deste trabalho é que não foi conseguida a coordenação necessária entre as várias tarefas de execução para que todos os requisitos necessários, deste modo de poupança de energia, fossem atingidos.

Para comparação é seguidamente apresentado o gráfico obtido quando é usada a implementação padrão existente na plataforma SunSPOT. Observa-se que apresenta uma maior percentagem de tempo em modo ativo do que o MH-MAC, existindo o potêncial para poupar energia. No entanto, atendendo a que a poupança energética é maior no estado de sono profundo, verifica-se que o

protótipo atual do MH-MAC ainda é menos eficiente do que o protocolo MAC existente por omissão nos SunSPOT.

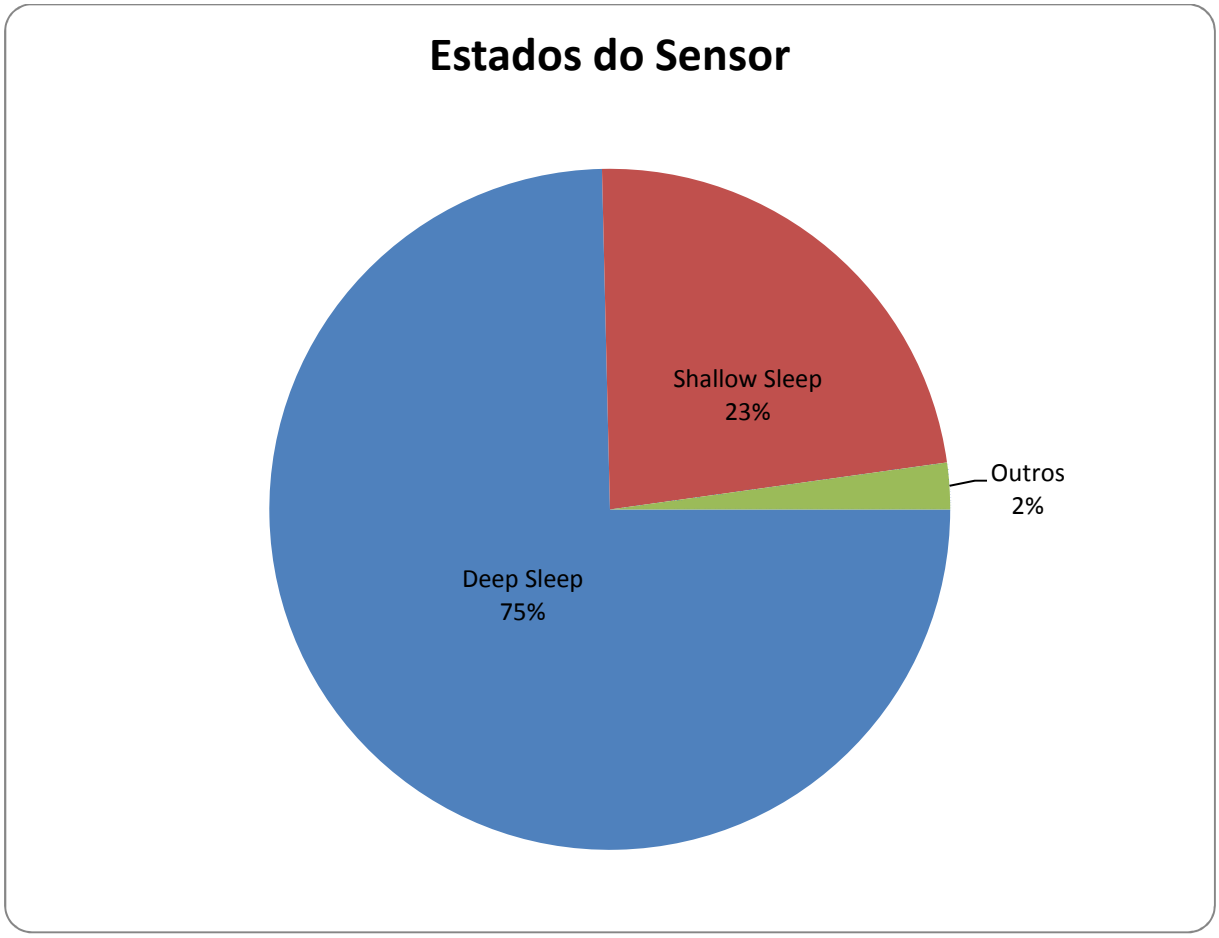

Figura 4.4 – Gráfico dos Estados do Sensor para a implemantação padão

4.3.3. Bateria

O gráfico seguinte representa a evolução do nível voltagem da bateria do sensor ao longo do tempo.

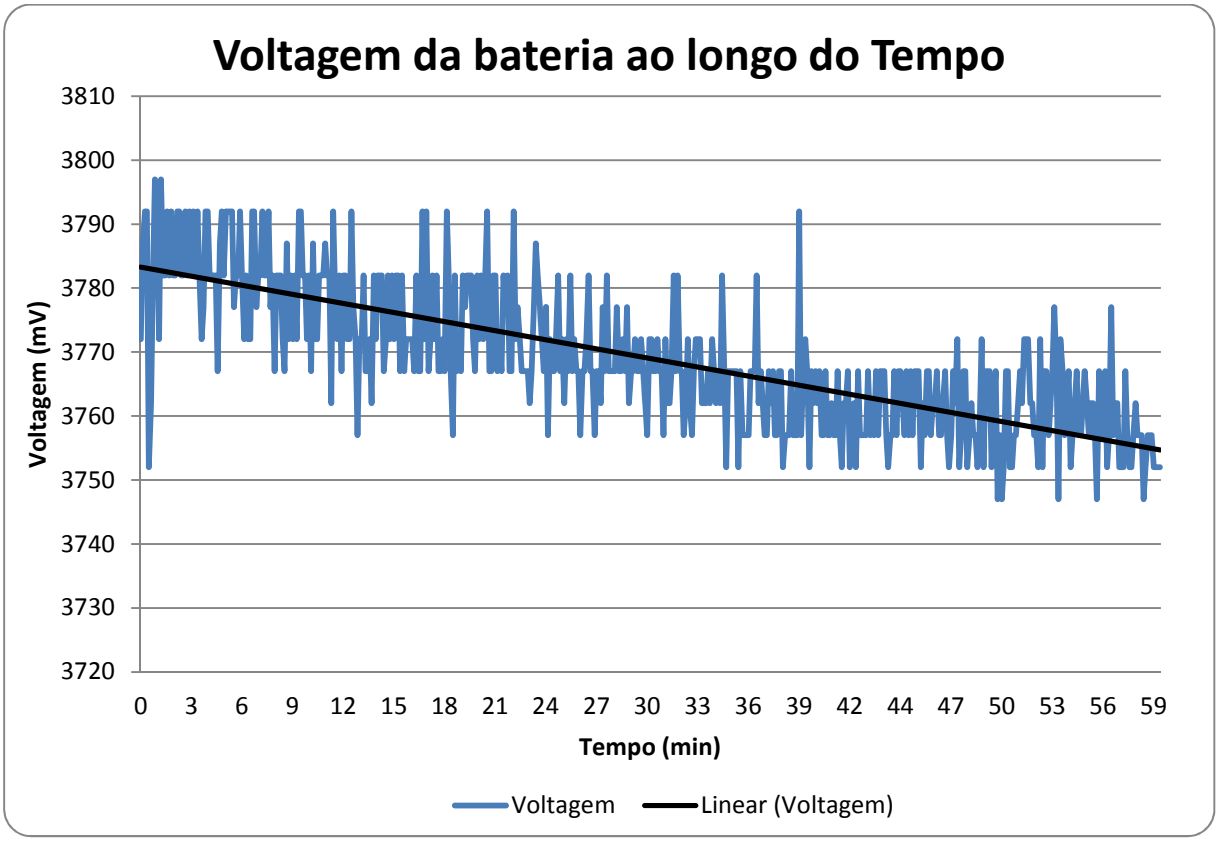

Figura 4.5 – Gráfico da Bateria ao longo do Tempo

Como se pode observar pelo gráfico, o sensor apresentou um consumo médio de 0.02Ah. Este valor está dentro dos parâmetros definidos pelo manual dos SunSPOT, mas muito longe dos valores que seria possível atingir caso o sensor entrasse no estado de maior poupança de energia, que rondaria o valor de 35µA.

4.4. Comparação com outros Sensores

Os sensores SunSPOT são equipados com processadores mais potentes (32 bits) do que a maior parte dos outros sensores sem fios (tipicamente com processadores de 8 bits) e também têm muito mais memória. Pretende-se aqui comparar sumariamente os consumos energéticos dos SunSPOT com outros sensores TelosB a correr o mesmo protocolo MH-MAC em TinyOS. Como termo de comparação foram usados os dados de desempenho apresentados no artigo [22] e numa tese de mestrado [23].

Seguidamente apresenta-se uma tabela com os valores apresentados pelos dois sensores. Depois de analisados os dados, é possível observar que os sensores TelosB a correr TinyOS, tal como era esperado, são bastante mais conservadores em termos de energia, pois apresentam consumos mais baixos nos vários modos de operação.

Tabela 4.1 - Comparação dos consumos

	SunSPOT	**TelosB (TinyOS)**
Modo ativo (mA)	70 a 120	27,85
Modo intermédio (mA)	24	6,35
Modo de poupança (µA)	32	14

Capítulo 5. Conclusões

5.1. Síntese Geral

O objectivo desta dissertação era o de desenvolver o protocolo MMH-MAC numa plataforma recente de sensores, a plataforma SunSPOT. Como já mencionado acima este objectivo não foi atingido, não tendo sido possível a total implementação do protocolo.

Foram também apontadas algumas deficiências da plataforma escolhida e a solução para ultrapassar algumas dessas deficiências. Essa solução consistiu na realização d modo assíncrono do MMH-MAC sobre um módulo de desenvolvimento de protocolos MAC, AnyMAC, realizado na plataforma SunSPOT.

Depois foi apresentada a descrição da solução implementada com base em AnyMAC, os resultados dessa implementação e o cenário usado na colecta de valores.

Por fim, os sensores da plataforma SunSPOT foram comparados com os TinyOS

5.2. Conclusões

Com a execução deste trabalho podemos concluir que a plataforma SunSPOT apresenta uma interface agradável a quem a quiser utilizar para realizar aplicações. Isto pode ser observado em vários pontos começando pela linguagem utilizada, o Java e continuando com a inclusão de vários manuais e exemplos. Está também presente e completamente integrado na plataforma um simulador. Este simulador é bastante extenso sendo capaz de simular todos os aspetos dos sensores reais. Outro dos pontos fortes desta plataforma é a disponibilização de todo o código fonte.

Uma das deficiências desta plataforma é a de a sua implementação estar optimizada para um protocolo MAC específico, sendo por isso difícil a introdução de novos protocolos. Nesta dissertação procurou-se usar a plataforma AnyMAC para realizar o desenvolvimento de um protocolo MAC alternativo. No entanto, esta tarefa revelou-se demasiado complexa. Apesar do protocolo ter sido parcialmente realizado, não foi possível libertar todas as tarefas no sistema de maneira a deixar o sensor sem fios entrar em modo de sono profundo. Conclui-se, assim, que a plataforma AnyMAC não é atualmente uma alternativa viável para a realização de protocolos MAC mais complexos, devido entre outros pontos, à falta de documentação.

Quando se comparam os sensores SunSPOT com os TelosB (a correr TinyOS), é possível observar que os SunSPOT apresentam um maior consumo energético. Isto deve-se às significativas diferenças de *hardware* entre os dois dispositivos, onde o SunSPOT apresenta um processador de muito maior capacidade e mais memória. A máquina virtual Java providencia maior facilidade em termos de desenvolvimento de aplicações, mas gasta substancialmente mais energia quando está em execução, tornando este tipo de sensores viável apenas quando existem fontes de energia disponíveis.

5.3. Trabalho Futuro

Durante o desenvolvimento deste trabalho foram observados vários pontos que podiam ser melhorados ou modificados. Em primeiro lugar os simuladores podiam ser aumentados para suportar o AnyMAC. Isto faria com que o desenvolvimento se tornasse muito mais simples em termos de depuração de erros.

Em segundo ponto, o uso do AnyMAC foi necessário devido a algumas deficiências da plataforma SunSPOT em termos de desenvolvimento de outros protocolos de controlo de acesso ao meio. Apesar do AnyMAC ser uma solução para esse problema talvez não seja a escolha indicada para todos os tipos de protocolos, sendo isto visível nas dificuldades encontradas na realização deste trabalho. Seria desejável estudar soluções alternativas e fazer uma comparação.

Por último, seria interessante integrar numa única rede sensores diferentes tais como SunSPOT e outros sensores de menor consumo energético, como os TelosB ou IRIS. Desta forma, seria possível combinar a superior capacidade de processamento dos SunSPOT com a maior eficiência energética dos TelosB ou IRIS.

Capítulo 6. Bibliografia

[1] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal, "Wireless sensor network survey," *Computer Networks*, vol. 52, no. 12, p. 39, Abril 2008.

[2] A. Willig H. Karl, *Protocols and Architectures for Wireless Sensor Networks*.: John Wiley & Sons, 2005.

[3] H.A.B.D. Oliveira, E.F. Nakamura e A.A.F. Loureiro A. Boukerche, "Localization Systems for Wireless Sensor Networks," *IEEE Wireless Communications*, vol. 14, no. 6, Dezembro 2007.

[4] N. Kushalnagar, J. Hui e D. Culler G. Montenegro, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," *IETF RFC 4944*, Setembro 2007.

[5] K. Jamieson, and H. Balakrishnan Y. C. Tay, "Collision-Minimizing CSMA and Its Applications to Wireless Sensor Networks," *IEEE J Sel. Areas Commun*, vol. 22, no. 6, 2004.

[6] A. Woo and D. Culler, "A Transmission Control Scheme for Media," in *ACM Mobicom*, Rome, July 2001.

[7] S. Singh and C. S. Raghavendra, "PAMAS - power aware multi-access protocol with signaling for ad hoc networks," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 3, 1998.

[8] K. Obraczka, and J. Garcia-Luna-Aceves V. Rajendran, "Energy-Efficient, Collision-Free Medium Access Control for Wireless Sensor Networks," in *SenSys*, Los Angeles, Novembro 2003.

[9] J. Garcia-Luna-Aceves, and K. Obraczka V. Rajendran, "Energy-Efficient, Application-Aware Medium Access for Sensor Networks," in *Mobile Adhoc and Sensor Systems Conference*, Novembro 2005.

[10] U. Roedig, and C. Sreenan A. Barroso, "uMAC: an energy-efficient medium access control for wireless sensor networks," in *2nd IEEE European Workshop on Wireless Sensor Networks*, Fevereiro 2005.

[11] J. Heidemann, and D. Estrin W. Ye, "An energy-efficient MAC protocol for wireless sensor networks," *Annual Joint Conference of the Computer and Communication Societies*, vol. 3, 2002.

[12] T. Van Dam and K. Langendoen, "An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks," in *SenSys*, Los Angeles, Novembro 2003.

[13] H. Pham and S. Jha, "An Adaptive Mobility-Aware MAC Protocol for Sensor Networks (MS-MAC)," Outubro 2004.

[14] J. Hill, and D. Culler J. Polastre, "Versatile Low Power Media Access for Wireless Sensor Networks," in *SenSys*, Baltimore, Novembro 2004.

[15] E. Yee, Gary V.and Anderson, and R. Han M. Buettner, "X-MAC: A Short Preamble MAC Protocol for Duty-Cycled Wireless Sensor Networks," Technical Report CU-CS-1008-06 Maio 2006.

[16] J.-D. Decotignie, C. Enz, and E. Le Roux A. El-Hoiydi, "Poster Abstract: WiseMac, an Ultra Low Power MAC Protocol for the WiseNET Wireless Sensor Networks," in *SenSys*, Los Angeles, Novembro 2003.

[17] A. Warrier, M. Aia, and J. Min I. Rhee, "ZMAC: a Hybrid MAC for Wireless Sensor Networks," in *SenSys*, San Diego, Novembro 2005.

[18] R. Oliveira, M. Pereira, M. Macedo, and P. Pinto L. Bernardo, "A Wireless Sensor MAC Protocol for Bursty Data Traffic," *IEEE PIMRC*, 2007.

[19] E. Miluzzo, S. G. Campbell, Andrew T.and Hong, and F. Cuomo G.-S. Ahn, "Funneling-MAC: A Localized, Sink-Oriented MAC For Boosting Fidelity in Sensor Networks," in *SenSys*, Boulder, Novembro 2006.

[20] Abdelmalik Bachir, Mischa Dohler, Thomas Watteyne, and Kin K. Leung, "MAC Essentials for Wireless Sensor Networks," *IEEE COMMUNICATIONS SURVEYS & TUTORIALS*, vol. 12, no. 2, 2010.

[21] Daniel van den Akker. (2009) AnyMAC. [Online]. http://www.pats.ua.ac.be/content/software/anymac/index.php/Welcome

[22] Luís Bernardo et al., "A MAC protocol for mobile wireless sensor networks with bursty traffic," in *Wireless Communications and Networking Conference (WCNC)*, Sydney, Setembro 2010.

[23] Hugo Manuel Serrão Borda d' Água, Protocolo MAC para acesso multi-modo em redes de sensores sem fios móveis, Dissertação de Mestrado em Engenharia Electrotécnica e de Computadores, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2008.

Apêndice A. MHMAC.java

```
package be.ac.ua.pats.anymac.mhmac;

import java.io.IOException;

import be.ac.ua.pats.anymac.hal.CoreComponent;
import be.ac.ua.pats.anymac.hal.IHardwareAbstractionLayer;
import be.ac.ua.pats.anymac.hal.components.ChannelController;
import be.ac.ua.pats.anymac.hal.components.PowerController;
import be.ac.ua.pats.anymac.mac.AddressTranslator;
import be.ac.ua.pats.anymac.mac.GenericMACLayer;
import be.ac.ua.pats.anymac.mac.MACPacket;
import be.ac.ua.pats.anymac.mac.MACPacketFactory;
import be.ac.ua.pats.anymac.mac.MACSettingsManager;

import com.sun.spot.peripheral.ChannelBusyException;
import com.sun.spot.peripheral.ILed;
import com.sun.spot.peripheral.NoAckException;
import com.sun.spot.peripheral.Spot;
import com.sun.spot.peripheral.SpotFatalException;
import com.sun.spot.util.Queue;
import com.sun.spot.util.Utils;

public class MHMAC extends GenericMACLayer {

  /**
   * The last preamble packet group number.
   */
  private byte preambleGroupNumber;

  /**
   * The last data packet group number.
   */
  private byte dataGroupNumber;

  /**
   * Led used to signal packet transmission
   */
  private ILed sendLed = Spot.getInstance().getRedLed();

  /**
   * Led used to signal packet reception
   */
  private ILed receiveLed = Spot.getInstance().getGreenLed();

  /**
   * Whether to show usage by blinking LEDs
   */
  private boolean showUse = true;

  /**
   * The maximum number of times the a CSMA send may be attempted before giving up
   */
  private static final int MAX_SEND_RETRIES = 3;

  /**
   * The number of millis to wait between retries
   */
```

```
  private static int[] RETRY_WAITS;

  {
    RETRY_WAITS = new int[MAX_SEND_RETRIES];
    RETRY_WAITS[0] = 0;
    RETRY_WAITS[1] = 2000;
    for (int i = 2; i < RETRY_WAITS.length; i++) {
      RETRY_WAITS[i] = 4000;
    }
  }

  /**
   * The maximum time to wait for an acknowledgement
   */
  private static final int ACK_TIMEOUT = 1000;

  private static final int PREAMBLE_ACK_TIMEOUT = 100;

  /**
   * This class is used to insert received unicast packet into the packetQueue only AFTER the
acknowledgement has been
   * sent. This helps in preventing NoAckExceptions
   *
   * @author Daniel van den Akker
   */
  private class AckSentListener implements MHMACPacketTransmissionListener {

    /**
     * The packet to be added to the reception queue
     */
    private MHMACPacket packet;

    /**
     * @param packet - the packet to be added to the queue
     */
    public AckSentListener(MHMACPacket packet) {
      this.packet = packet;
    }

    public void sendDone(MHMACPacket packet, boolean success) {
      if (success) {
        packetQueue.put(this.packet);
      }
    }

  }

  private class DataPacketListener implements MHMACPacketReceptionListener {

    public void receive(MHMACPacket packet) {
      if (!packet.getDestinationAddress().equals(MHMACAddress.BROADCAST_ADDRESS)) {
        MHMACPacket ackPacket = new MHMACPacket(MHMACPacket.ACK_TYPE);
        ackPacket.setPayloadLength(0);
        ackPacket.setSourceAddress(addressTranslator.getDeviceAddress());
        ackPacket.setDestinationAddress(packet.getSourceAddress());
        ackPacket.setSequenceNumber(packet.getSequenceNumber());
        packetSender.enqueuePacket(ackPacket, new AckSentListener(packet));
      } else {
        packetQueue.put(packet);
      }
```

```
    }

  }

  private class AckPacketListener implements MHMACPacketReceptionListener {

    /**
     * The object threads synchronize on in order to access the last seqNr received by ack packet
     */
    private Object ackMonitor = new String();

    /**
     * The most recent AckPacket
     */
    private MHMACPacket lastAckPacket;

    private boolean waitingForAck = false;

    public void receive(MHMACPacket packet) {
      //we received an ackpacket
      synchronized (ackMonitor) {
        //if we're waiting for an ack: update lastAckPacket and notify the waiter
        if (waitingForAck) {
          lastAckPacket = packet;
          waitingForAck = false;
          ackMonitor.notify();
        }
      }
    }

    /**
     * Wait until the ack for a specific packet has arrived
     *
     * @param seqNr the expected sequence number
     * @return whether the ack was received or not
     */
    public boolean waitForAck(byte seqNr) {
      for (long startTime = System.currentTimeMillis(); System.currentTimeMillis() < startTime +
ACK_TIMEOUT;) {
        MHMACPacket ackPacket;
        synchronized (ackMonitor) {
          if (lastAckPacket != null) {
            throw new SpotFatalException("ACK already there when about to wait for it");
          }
          waitingForAck = true;
          try {
            ackMonitor.wait(ACK_TIMEOUT);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
          waitingForAck = false;
          ackPacket = lastAckPacket;
          lastAckPacket = null;
        }

        if (ackPacket == null) {
          return false;
        } else if (ackPacket.getSequenceNumber() == seqNr) {
          return true;
        }
```

```
      }
      return false;
    }

  }

  private class PreamblePacketListener implements MHMACPacketReceptionListener {

    public void receive(MHMACPacket packet) {
      final int minusTime = 10;
      if (packet.getGroupNumber() != preambleGroupNumber) {
        preambleGroupNumber = packet.getGroupNumber();
        if (!packet.getDestinationAddress().equals(MHMACAddress.BROADCAST_ADDRESS)) {
          MHMACPacket ackPacket = new MHMACPacket(MHMACPacket.PREAMBLE_ACK_TYPE);
          ackPacket.setPayloadLength(0);
          ackPacket.setSourceAddress(addressTranslator.getDeviceAddress());
          ackPacket.setDestinationAddress(packet.getSourceAddress());
          ackPacket.setSequenceNumber(packet.getSequenceNumber());
          packetSender.enqueuePacket(ackPacket, new AckSentListener(packet));
        } else {
          long timeToSend = packet.getPayload().readLongAt(0);
          //System.out.println("Going to sleep for: " + (timeToSend - minusTime));
          // The timeToData value was taken some time ago so we sleep the value minus a litle bit.
          // TODO: check the litle bit.
          Utils.sleep(timeToSend - minusTime);
        }
      }
    }

  }

// TODO: Maybe merge this class with AckPacketListener
  private class PreambleAckPacketListener implements MHMACPacketReceptionListener {

    /**
     * The object threads synchronize on in order to access the last seqNr received by ack packet
     */
    private Object ackMonitor = new String();

    /**
     * The most recent AckPacket
     */
    private MHMACPacket lastAckPacket;

    private boolean waitingForAck = false;

    public void receive(MHMACPacket packet) {
      //we received an ackpacket
      synchronized (ackMonitor) {
        //if we're waiting for an ack: update lastAckPacket and notify the waiter
        if (waitingForAck) {
          lastAckPacket = packet;
          waitingForAck = false;
          ackMonitor.notify();
        }
      }
    }

    /**
     * Wait until the ack for a specific packet has arrived
```

```
     *
     * @param seqNr the expected sequence number
     * @return - whether the ack was received or not
     */
    public boolean waitForAck(byte seqNr) {
      for (long startTime = System.currentTimeMillis(); System.currentTimeMillis() < startTime +
PREAMBLE_ACK_TIMEOUT;) {
        MHMACPacket ackPacket;
        synchronized (ackMonitor) {
          if (lastAckPacket != null) {
            throw new SpotFatalException("ACK already there when about to wait for it");
          }
          waitingForAck = true;
          try {
            ackMonitor.wait(PREAMBLE_ACK_TIMEOUT);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
          waitingForAck = false;
          ackPacket = lastAckPacket;
          lastAckPacket = null;
        }

        if (ackPacket == null) {
          return false;
        } else if (ackPacket.getSequenceNumber() == seqNr) {
          return true;
        }
      }
      return false;
    }

  }

  /**
   * The packet factory
   */
  private MHMACPacketFactory packetFactory;

  /**
   * The address translator
   */
  private MHMACAddressTranslator addressTranslator;

  /**
   * The settings manager
   */
  private MHMACSettingsManager settingsManager;

  /**
   * The class responsible for receiving packets from the hal
   * */
  private MHMACPacketReceiver receiver;

  /**
   * The thread in which the receiver will run
   * */
  private Thread receiverThread;

  /**
```

```
 * The current sequence number
 */
private volatile byte currentSeqNr = (byte) System.currentTimeMillis();

/**
 * The current group sequence number
 */
private volatile byte groupSeqNr = (byte) System.currentTimeMillis();

/**
 * the hardware abstraction layer
 */
private IHardwareAbstractionLayer layer;

/**
 * the core component of the hardware abstraction layer
 */
private CoreComponent core;

/**
 * The queue for received packets
 */
private Queue packetQueue;

/**
 * Whether the MAC Layer is enabled
 */
private boolean enabled;

/**
 * The ack packet listener
 */
private AckPacketListener ackListener;

/**
 * The data packet listener
 */
private DataPacketListener dataListener;

private PreamblePacketListener preambleListener;

private PreambleAckPacketListener preambleAckListener;

/**
 * The MAC component in charge of putting packets on the link
 */
private MHMACPacketSender packetSender;

/**
 * The thread in which the packetSender is executed
 */
private Thread packetSenderThread;

public MACPacketFactory getPacketFactory() {
  return packetFactory;
}

public MACSettingsManager getSettingsManager() {
  return settingsManager;
}
```

```
  public AddressTranslator getTranslator() {
    return addressTranslator;
  }

  public void initialize(IHardwareAbstractionLayer hal) {
    layer = hal;
    core = (CoreComponent) layer.getComponent(CoreComponent.class);
    settingsManager = new MHMACSettingsManager((PowerController)
layer.getComponent(PowerController.class),
        (ChannelController) layer.getComponent(ChannelController.class));
    //HWAutoAckControl
    packetFactory = new MHMACPacketFactory();

    packetQueue = new Queue();
    ackListener = new AckPacketListener();
    dataListener = new DataPacketListener();
    // Preamble listeners
    preambleListener = new PreamblePacketListener();
    preambleAckListener = new PreambleAckPacketListener();

    enabled = false;

    packetSender = new MHMACPacketSender(core);
    packetSenderThread = new Thread(packetSender);
    packetSenderThread.setPriority(Thread.MAX_PRIORITY);

    receiver = new MHMACPacketReceiver(core);
    receiver.setEnabled(false);
    receiver.addListener(MHMACPacket.ACK_TYPE, ackListener);
    receiver.addListener(MHMACPacket.DATA_TYPE, dataListener);
    // Preamble listeners
    receiver.addListener(MHMACPacket.PREAMBLE_TYPE, preambleListener);
    receiver.addListener(MHMACPacket.PREAMBLE_ACK_TYPE, preambleAckListener);

    addressTranslator = new MHMACAddressTranslator(this);

    receiverThread = new Thread(receiver, "MAC Packet receiver thread");
    receiverThread.setPriority(Thread.MAX_PRIORITY - 1);

    receiverThread.start();
    packetSenderThread.start();

  }

  public synchronized boolean isReceiverEnabled() {
    return enabled;
  }

  /**
   * the 'last packet received' queue
   */
  private int[] lastPackets = new int[10];

  {
    for (int i = 0; i < lastPackets.length; i++) {
      lastPackets[i] = 0xFFFFFF;
    }
  }
```

```
  /**
   * where in array the information of the next received packet is written
   */
  private int head = 0;

  public MACPacket receivePacket() throws IOException {
    if (showUse) {
      receiveLed.setOn(true);
    }

    /*
     * The use of software acknowledgements sometimes causes packets to be received twice we
therefore use some sort
     * of 'round-robin' buffer containing the address and the seq. nr of the last few packets that arrived.
     * Each received packet is then checked against this history and the double packets are filtered out
     * */
    MHMACPacket retVal = null;
    while (retVal == null) {
      MHMACPacket packet = (MHMACPacket) packetQueue.get();
      if (packet.getDestinationAddress().equals(MHMACAddress.BROADCAST_ADDRESS)) {
        retVal = packet;
        break;
      }
      synchronized (lastPackets) {
        boolean found = false;
        int hash = (int) (0 | (((((MHMACAddress) packet.getSourceAddress()).asLong() & 0xFFFF) <<
8) | (packet.
            getSequenceNumber() & 0xFF)));
        for (int i = 0; i < lastPackets.length; i++) {
          if (hash == lastPackets[i]) {
            found = true;
            break;
          }
        }
        if (!found) {
          lastPackets[head] = hash;
          head = (head + 1) % lastPackets.length;
          retVal = packet;
          break;
        }
      }
    }
    if (showUse) {
      receiveLed.setOn(false);
    }
    return retVal;

  }

  public Class[] requiredHALComponents() {
    return new Class[]{PowerController.class, ChannelController.class};
  }

  public synchronized void sendPacket(MACPacket packet) throws IOException {
    if (showUse) {
      sendLed.setOn(true);
    }

    if (!isReceiverEnabled()) {
      receiver.setEnabled(true);
```

```
    }
    try {
      MHMACPacket pkt = (MHMACPacket) packet;
      pkt.setSequenceNumber(currentSeqNr++);
      subSend(pkt);
    } finally {
      receiver.setEnabled(this.isReceiverEnabled());
    }
    if (showUse) {
      sendLed.setOn(false);
    }
  }

  public synchronized void setReceiverEnabled(boolean enabled) {
    if (this.enabled != enabled) {
      receiver.setEnabled(enabled);
      this.enabled = enabled;
    }
  }

  /**
   * try to send a packet
   *
   * @param packet the packet to be sent
   * @throws ChannelBusyException if the channel was busy
   * @throws NoAckException if an ack could not be timely received
   */
  private void subSend(MHMACPacket packet) throws ChannelBusyException, NoAckException {
    for (int i = 0; i < MAX_SEND_RETRIES; i++) {
      boolean result = doSingleSend(packet);
      if (!result) {
        throw new ChannelBusyException("Failed to send Packet.");
      } else if (packet.getDestinationAddress().equals(MHMACAddress.BROADCAST_ADDRESS)) {
        return;
      } else if (ackListener.waitForAck(packet.getSequenceNumber())) {
        return;
      } else {
        if (i < MAX_SEND_RETRIES - 1) {
          Utils.sleep(RETRY_WAITS[i]);
        }
      }
    }
    throw new NoAckException("Failed to Receive acknowledgement for sent packet: " +
packet.getSequenceNumber());


  }

  private class SendListener implements MHMACPacketTransmissionListener {

    /**
     * Whether the packet was sent successfully. Null if the packet has not yet been processed
     */
    Boolean success = null;

    public synchronized void sendDone(MHMACPacket packet, boolean success) {
      this.success = new Boolean(success);
      this.notifyAll();
    }
```

```
    /**
     * Waits until a packet has been sent
     *
     * @return - whether the packet was sent sucessfully
     */
    public synchronized boolean waitUntilSent() {
      while (success == null) {
        try {
          this.wait();
        } catch (InterruptedException e) {
        }
      }
      return success.booleanValue();
    }

  }

  /**
   * Puts a single packet on the link, waiting until the packet has actually been sent.
   *
   * @param packet the packet to be sent
   * @return whether the send was successfull
   */
  boolean doSingleSend(MHMACPacket packet) throws ChannelBusyException {
    long twiceDutyCicle = 2 * 10 * 11;
    long elapsedTime = 0;
    long startTime = System.currentTimeMillis();
    while (twiceDutyCicle > elapsedTime) {
      long timeToData = twiceDutyCicle - elapsedTime;
//                        System.out.println("Total Time: " + twiceDutyCicle
//                                + "; Elapsed Time: " + elapsedTime
//                                + "; Time to data: " + timeToData);

      MHMACPacket preamble = new MHMACPacket(MHMACPacket.PREAMBLE_TYPE);
      preamble.setSourceAddress(MHMACAddress.SPOT_ADDRESS);
      preamble.getPayload().writeLongAt(0, timeToData);
      preamble.setDestinationAddress(packet.getDestinationAddress());
      preamble.setPayloadLength(Utils.SIZE_OF_LONG);
      preamble.setGroupNumber(groupSeqNr);

      SendListener listener = new SendListener();
      packetSender.enqueuePacket(preamble, listener);
      boolean result = listener.waitUntilSent();

      if (!result) {
        throw new ChannelBusyException("Failed to send Packet.");
      } else if (packet.getDestinationAddress().equals(MHMACAddress.BROADCAST_ADDRESS)) {
        // Trying to do broadcast so full preamble is needed.
        Utils.sleep(10);
        elapsedTime = System.currentTimeMillis() - startTime;
        continue;
      } else if (preambleAckListener.waitForAck(preamble.getSequenceNumber())) {
        // Trying to do unicast so we can skip the rest of the preambles as soon as the receiver is
ready
        break;
      }
      // Doing broadcast or no preamble received
      elapsedTime = System.currentTimeMillis() - startTime;
    }
```

```
      SendListener listener = new SendListener();
      // Update group number
      packet.setGroupNumber(groupSeqNr++);
      packetSender.enqueuePacket(packet, listener);
      return listener.waitUntilSent();
    }

    /**
     * @return - the packetreceiver
     */
    MHMACPacketReceiver getReceiver() {
      return receiver;
    }

}
```

Apêndice B. MHMACAddress.java

```
package be.ac.ua.pats.anymac.mhmac;

import be.ac.ua.pats.anymac.mac.MACAddress;
import be.ac.ua.pats.anymac.net.NetAddress;

import com.sun.spot.util.Utils;

/**
 * The MAC Address class for the {@link MHMAC} implementation.
 */
public class MHMACAddress extends MACAddress {

  /**
   * The MAC Broadacast address.
   */
  public static final MHMACAddress BROADCAST_ADDRESS = new MHMACAddress(-1L);

  /**
   * The Device MAC Address of this SunSPOT.
   */
  public static final MHMACAddress SPOT_ADDRESS = new
MHMACAddress(NetAddress.getSpotAddress().asLong());

  /**
   * The address itself.
   */
  private long address;

  /**
   * Package constructor.
   *
   * @param address the long representation of the address
   */
  MHMACAddress(long address) {
    this.address = address;
  }

  public boolean equals(MACAddress address) {
    return (address instanceof MHMACAddress) && ((MHMACAddress) address).address ==
this.address;
  }

  public void fromByteArray(byte[] buf) {
    this.address = Utils.readLittleEndLong(buf, 0);
  }

  public boolean equals(Object obj) {
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    final MHMACAddress other = (MHMACAddress) obj;
    if (this.address != other.address) {
      return false;
    }
```

```
      return true;
    }

    public int hashCode() {
      return (int) (address & 0xFFFF);
    }

    public byte[] toByteArray() {
      byte[] arr = new byte[8];
      Utils.writeLittleEndLong(arr, 0, address);
      return arr;
    }

    public String toString() {
      return Long.toString(address & 0xFFFF);
    }

    /**
     * Return the address as a long.
     * @return the long representation of the address
     */
    long asLong() {
      return address;
    }

}
```

Apêndice C. MHMACAddressTranslator.java

```
package be.ac.ua.pats.anymac.mhmac;

import java.io.IOException;
import java.util.Hashtable;

import be.ac.ua.pats.anymac.mac.AddressTranslator;
import be.ac.ua.pats.anymac.mac.MACAddress;
import be.ac.ua.pats.anymac.net.NetAddress;

/**
 * The {@link AddressTranslator} for the {@link MHMAC}.
 */
public class MHMACAddressTranslator implements AddressTranslator {

  /**
   * The NetAddress -> MACAddress lookup table.
   */
  private Hashtable netToMACHash;

  /**
   * The MACAddress -> NetAddress lookup table.
   */
  private Hashtable macToNetHash;

  /**
   * Constructor.
   *
   * @param macLayer the MACLayer being used
   */
  public MHMACAddressTranslator(MHMAC macLayer) {
    netToMACHash = new Hashtable();
    macToNetHash = new Hashtable();

    netToMACHash.put(NetAddress.getSpotAddress(), getDeviceAddress());
    netToMACHash.put(NetAddress.getBroadcastAddress(), getBroadcastAddress());

    macToNetHash.put(getDeviceAddress(), NetAddress.getSpotAddress());
    macToNetHash.put(getBroadcastAddress(), NetAddress.getBroadcastAddress());
  }

  public MACAddress fromByteArray(byte[] inputBuffer) {
    MHMACAddress address = new MHMACAddress((short) 0);
    address.fromByteArray(inputBuffer);
    return address;
  }

  public final MACAddress getBroadcastAddress() {
    return MHMACAddress.BROADCAST_ADDRESS;
  }

  public final MACAddress getDeviceAddress() {
    return MHMACAddress.SPOT_ADDRESS;
  }

  public MACAddress translate(NetAddress address) throws IOException {
    if (address.equals(NetAddress.getBroadcastAddress())) {
      return MHMACAddress.BROADCAST_ADDRESS;
```

```
    } else if (address.equals(NetAddress.getSpotAddress())) {
      return MHMACAddress.SPOT_ADDRESS;
    } else {
      MHMACAddress mac = new MHMACAddress(address.asLong());
      netToMACHash.put(address, mac);
      return mac;
    }
  }

  public NetAddress translate(MACAddress address) throws IOException {
    if (address.equals(MHMACAddress.BROADCAST_ADDRESS)) {
      return NetAddress.getBroadcastAddress();
    } else if (address.equals(MHMACAddress.SPOT_ADDRESS)) {
      return NetAddress.getSpotAddress();
    } else {
      NetAddress net = new NetAddress(((MHMACAddress) address).asLong());
      macToNetHash.put(address, net);
      return net;
    }
  }

  /**
   * Return the number of known hosts.
   * @return the number of known hosts
   */
  public int numberOfKnownHosts() {
    return macToNetHash.size();
  }

}
```

Apêndice D. MHMACPacket.java

```
package be.ac.ua.pats.anymac.mhmac;

import be.ac.ua.pats.anymac.IPacket;
import be.ac.ua.pats.anymac.PacketPayload;
import be.ac.ua.pats.anymac.hal.PHYPacket;
import be.ac.ua.pats.anymac.mac.MACAddress;
import be.ac.ua.pats.anymac.mac.MACHeaderInfo;
import be.ac.ua.pats.anymac.mac.components.AbstractMACPacket;

/**
 * The MACPacket class for the {@link MHMAC} implementation.
 */
public class MHMACPacket extends AbstractMACPacket {

   /**
    * Byte indicating a the packet is a datapacket.
    */
   static final byte DATA_TYPE = (byte) 0x0;

   /**
    * Byte indicating a the packet is an ack packet.
    */
   static final byte ACK_TYPE = (byte) 0x1;

   /**
    * Byte indicating the packet is an Preamble packet.
    */
   static final byte PREAMBLE_TYPE = (byte) 0x2;

   /**
    * Byte indicating the packet is an Preamble ack packet.
    */
   static final byte PREAMBLE_ACK_TYPE = (byte) 0x3;

   /**
    * The offset for the destination address field.
    */
   private static final int DESTINATION_ADDRESS_FIELD_OFFSET = 0;

   /**
    * The length of the destination address field.
    */
   private static final int DESTINATION_ADDRESS_FIELD_LENGTH = 8;

   /**
    * The offset for the source address field.
    */
   private static final int SOURCE_ADDRESS_FIELD_OFFSET =
DESTINATION_ADDRESS_FIELD_OFFSET + DESTINATION_ADDRESS_FIELD_LENGTH;

   /**
    * The length of the source address field.
    */
   private static final int SOURCE_ADDRESS_FIELD_LENGTH = 8;

   /**
    * The offset for the type field.
```

```
   */
  private static final int TYPE_FIELD_OFFSET = SOURCE_ADDRESS_FIELD_OFFSET +
SOURCE_ADDRESS_FIELD_LENGTH;

  /**
   * The length of the type field.
   */
  private static final int TYPE_FIELD_LENGTH = 1;

  /**
   * The offset for the sequence number field.
   */
  private static final int SEQUENCE_NR_FIELD_OFFSET = TYPE_FIELD_LENGTH +
TYPE_FIELD_OFFSET;

  /**
   * The size of the sequence number field.
   */
  private static final int SEQUENCE_NR_FIELD_SIZE = 1;

  /**
   * The offset for the group number field.
   */
  private static final int GROUP_NR_FIELD_OFFSET = SEQUENCE_NR_FIELD_OFFSET +
SEQUENCE_NR_FIELD_SIZE;

  /**
   * The size of the group number field.
   */
  private static final int GROUP_NR_FIELD_SIZE = 1;

  /**
   * The total header length.
   */
  private static final int HEADER_LENGTH = GROUP_NR_FIELD_OFFSET +
GROUP_NR_FIELD_SIZE;

  /**
   * The maximum payload size.
   */
  private static final byte MAX_PAYLOAD_SIZE = PHYPacket.DATA_BUFFER_SIZE -
HEADER_LENGTH;

  /**
   * The object controlling the access to the packet payload.
   * */
  private PacketPayload payload;

  /**
   * Creates a new {@link MHMACPacket} of the specified type.
   * @param type the type of the packet
   */
  MHMACPacket(byte type) {
    super(new PHYPacket());
    this.setType(type);
    setPayloadLength(0);
    payload = new PacketPayload(phyPacket.getPayload(), this, HEADER_LENGTH);
  }

  /**
```

```
   * Create a new {@link MHMACPacket} from an existing {@link PHYPacket}.
   * @param packet the PHYPacket
   */
  protected MHMACPacket(PHYPacket packet) {
    super(packet);
    payload = new PacketPayload(phyPacket.getPayload(), this, HEADER_LENGTH);
  }

  public MACAddress getDestinationAddress() {
    return new
MHMACAddress(phyPacket.getPayload().readLongAt(DESTINATION_ADDRESS_FIELD_OFFSET));
  }

  public MACHeaderInfo getHeaderInfo() {
    return new MACHeaderInfo();
  }

  public MACAddress getSourceAddress() {
    return new
MHMACAddress(phyPacket.getPayload().readLongAt(SOURCE_ADDRESS_FIELD_OFFSET));
  }

  public void setDestinationAddress(MACAddress address) {
    phyPacket.getPayload().writeLongAt(DESTINATION_ADDRESS_FIELD_OFFSET,
((MHMACAddress) address).asLong());
  }

  public void setSourceAddress(MACAddress address) {
    phyPacket.getPayload().writeLongAt(SOURCE_ADDRESS_FIELD_OFFSET, ((MHMACAddress)
address).asLong());
  }

  public int getMaxPayloadLength() {
    return MAX_PAYLOAD_SIZE;
  }

  public int getPayloadLength() {
    return phyPacket.getPayloadLength() - HEADER_LENGTH;
  }

  public void setPayloadLength(int length) {
    phyPacket.setPayloadLength(length + HEADER_LENGTH);
  }

  public IPacket copy() {
    return new MHMACPacket((PHYPacket) phyPacket.copy());
  }

  public PacketPayload getPayload() {
    return payload;
  }

  /**
   * Return the type of the packet.
   * @return the type of the packet
   */
  public byte getType() {
    return phyPacket.getPayload().readByteAt(TYPE_FIELD_OFFSET);
  }
```

```
  /**
   * Set the type of the packet.
   * @param type the type of the packet
   */
  private void setType(byte type) {
    phyPacket.getPayload().writeByteAt(TYPE_FIELD_OFFSET, type);
  }

  /**
   * Return the sequence number of the packet.
   * @return the sequence number of the packet
   */
  public byte getSequenceNumber() {
    return phyPacket.getPayload().readByteAt(SEQUENCE_NR_FIELD_OFFSET);
  }

  /**
   * Set the sequence number.
   * @param seqNr the sequance number
   */
  public void setSequenceNumber(byte seqNr) {
    phyPacket.getPayload().writeByteAt(SEQUENCE_NR_FIELD_OFFSET, seqNr);
  }

  /**
   * Return the group number of the packet.
   * @return the group number of the packet
   */
  public byte getGroupNumber() {
    return phyPacket.getPayload().readByteAt(GROUP_NR_FIELD_OFFSET);
  }

  /**
   * Set the group number.
   * @param groupNr the group number
   */
  public void setGroupNumber(byte groupNr) {
    phyPacket.getPayload().writeByteAt(GROUP_NR_FIELD_OFFSET, groupNr);
  }

}
```

Apêndice E. MHMACPacketFactory.java

```
package be.ac.ua.pats.anymac.mhmac;

import be.ac.ua.pats.anymac.mac.MACPacket;
import be.ac.ua.pats.anymac.mac.MACPacketFactory;

/**
 * The Packet Factory for the {@link MHMAC} implementation.
 */
public class MHMACPacketFactory implements MACPacketFactory {

   public MACPacket newBroadcastPacket() {
      MHMACPacket pkt = new MHMACPacket(MHMACPacket.DATA_TYPE);
      pkt.setDestinationAddress(MHMACAddress.BROADCAST_ADDRESS);
      return pkt;
   }

   public MACPacket newUnicastPacket() {
      return new MHMACPacket(MHMACPacket.DATA_TYPE);
   }

}
```

Apêndice F. MHMACPacketReceiver.java

```
package be.ac.ua.pats.anymac.mhmac;

import be.ac.ua.pats.anymac.hal.CoreComponent;
import be.ac.ua.pats.anymac.hal.PHYPacket;

import com.sun.squawk.util.SquawkVector;

/**
 *
 * This class is responsible for receiving packets from the HAL,
 * decoding them and propagating them to the propper protocol handler.
 */
public class MHMACPacketReceiver implements Runnable {

   /**
    * The HAL Core component.
    */
   private CoreComponent core;

   /**
    * The listeners for each packet type.
    */
   private SquawkVector[] listeners;

   /**
    * whether reception is enabled or not.
    */
   private boolean enabled;

   /**
    * Wether the PacketReceiver must end.
    */
   private boolean terminated;

   /**
    * Constructor.
    *
    * @param core the core component of the hardwareabstraction layer
    */
   public MHMACPacketReceiver(CoreComponent core) {
      this.core = core;
      this.enabled = false;
      this.terminated = false;
      this.listeners = new SquawkVector[256];
      for (int i = 0; i < listeners.length; i++) {
         listeners[i] = null;
      }
   }

   /**
    * Add a packet listener.
    *
    * @param packetType the type of packet the listener is interested in
    * @param listener the listener itself
```

```
 */
public void addListener(byte packetType, MHMACPacketReceptionListener listener) {
  if (listeners[packetType & 0xFF] == null) {
    listeners[packetType & 0xFF] = new SquawkVector();
  }
  listeners[packetType].addElement(listener);
}

/**
 * Removes a packet listener.
 *
 * @param packetType the type of packet the listener was interested in
 * @param listener the listener to be removed
 */
public void removeListener(byte packetType, MHMACPacketReceptionListener listener) {
  if (listeners[packetType & 0xFF] != null) {
    listeners[packetType & 0xFF].removeElement(listener);
  }
}

/**
 * Returns whether the packets are being received or not.
 * @return whether packets are being received or not
 */
public synchronized boolean isEnabled() {
  return enabled;
}

/**
 * Turn packet reception on or off.
 *
 * @param enabled enables \ disables the packet reception
 */
public synchronized void setEnabled(boolean enabled) {
  this.enabled = enabled;
  core.setEnabled(enabled);
  if (this.isEnabled()) {
    this.notifyAll();
  }
}

/**
 * Returns whether the PacketReceiver is sheduled to exit.
 * @return whether the PacketReceiver is sheduled to exit
 */
public synchronized boolean isTerminated() {
  return terminated;
}

/**
 * Terminate the packet receiver.
 */
public synchronized void Terminate() {
  this.terminated = true;
  this.notifyAll();
}

public void run() {
  while (!isTerminated()) {
    synchronized (this) {
```

```
          while (!isEnabled() && !isTerminated()) {
            core.setEnabled(false);
            try {
               this.wait();
            } catch (InterruptedException e) {
            }
          }

        }

        if (isTerminated()) {
          break;
        }
        core.setEnabled(true);
        PHYPacket phy = core.receivePacket();
        if (phy.getPHYMetaData().isValidCRC()) {
          MHMACPacket demoPacket = new MHMACPacket(phy);
          if (demoPacket.getDestinationAddress().equals(MHMACAddress.SPOT_ADDRESS)
            || demoPacket.getDestinationAddress().equals(MHMACAddress.BROADCAST_ADDRESS))
{
            if (listeners[demoPacket.getType() & 0xFF] != null) {
              for (int i = 0; i < listeners[demoPacket.getType() & 0xFF].size(); i++) {
                ((MHMACPacketReceptionListener) listeners[demoPacket.getType() &
0xFF].elementAt(i)).receive(
                    demoPacket);
              }
            }
          }
        }
      }

    }

}
```

Apêndice G. MHMACPacketReceptionListener.java

```
package be.ac.ua.pats.anymac.mhmac;

/**
 * This interface is used by the {@link MHMACPacketReceiver} notify MAC
 * components interested in a specific type of {@link MHMACPacket}
 * certain types of packet that arrive.
 */
public interface MHMACPacketReceptionListener {

   /**
    * Further process the received packet.
    * @param packet the received packet
    */
   public void receive(MHMACPacket packet);

}
```

Apêndice H. MHMACPacketSender.java

```
package be.ac.ua.pats.anymac.mhmac;

import java.util.Random;

import be.ac.ua.pats.anymac.hal.CoreComponent;

import com.sun.spot.peripheral.ChannelBusyException;
import com.sun.spot.util.Queue;
import com.sun.spot.util.Utils;

/**
 * The {@link MHMACPacketSender} is responsible for putting MAC packets onto the link.
 * This is done in a separate class and in a standalone thread in order provide priority
 * for Ack packets.
 */
public class MHMACPacketSender implements Runnable {

  /**
   * The maximum numer of steps the CSMA process may take before giving up.
   */
  private static final int MAX_BACKOFF_RETRIES = 4;

  /**
   * The random number generator used for CSMA.
   */
  private Random random;

  /**
   * The component of the HAL needed for sending packets.
   */
  private CoreComponent core;

  /**
   * A helper class. This class allows storing packet related information that is needed to send a
packet, and update
   * the listener afterwards.
   */
  private class PacketInfo {

    /**
     * Constructor.
     *
     * @param packet the packet to be sent
     * @param listener the listener to be updated once the packet has been set on the link
     */
    PacketInfo(MHMACPacket packet, MHMACPacketTransmissionListener listener) {
      this.listener = listener;
      this.packet = packet;
    }

    /**
     * The packet to be sent.
     */
    public MHMACPacket packet;
```

```
    /**
     * The listener interested in the packet.
     */
    public MHMACPacketTransmissionListener listener;

  }

  /**
   * The queue of acks waiting to be sent. Packets from this Queue have priority over packets from the
data queue.
   */
  private Queue waitingAckPackets;

  /**
   * The queue of data packets waiting to be sent.
   */
  private Queue waitingDataPackets;

  /**
   * The monitor used to let the send thread sleep if no packets need to be sent.
   */
  private Object waitMonitor;

  /**
   * Used to allow the sender thread to be terminated.
   */
  private volatile boolean terminated;

  /**
   * Constructor.
   *
   * @param core the core component of the HAL
   */
  public MHMACPacketSender(CoreComponent core) {
    this.core = core;
    this.waitingAckPackets = new Queue();
    this.waitingDataPackets = new Queue();
    this.waitMonitor = new String();
    this.terminated = false;
    this.random = new Random(System.currentTimeMillis());
  }

  /**
   * Terminates the sender thread.
   */
  public void Terminate() {
    terminated = true;
    synchronized (waitMonitor) {
      waitMonitor.notify();
    }
  }

  /**
   * Returns whether the sender thread is terminated.
   * @return - whether the sender thread is terminated
   */
  public boolean isTerminated() {
    return terminated;
  }
```

```
public void enqueuePacket(MHMACPacket packet, MHMACPacketTransmissionListener listener) {
  if (packet.getType() == MHMACPacket.ACK_TYPE) {
    waitingAckPackets.put(new PacketInfo(packet, listener));
  } else {
    waitingDataPackets.put(new PacketInfo(packet, listener));
  }
  synchronized (waitMonitor) {
    waitMonitor.notify();
  }
}

public void run() {
  while (!isTerminated()) {
    synchronized (waitMonitor) {
      while (waitingAckPackets.isEmpty() && waitingDataPackets.isEmpty()) {
        try {
          waitMonitor.wait();
        } catch (InterruptedException e) {
        }
      }
    }

    while (!(waitingAckPackets.isEmpty() && waitingDataPackets.isEmpty())) {
      PacketInfo packet = null;
      synchronized (waitingAckPackets) {
        if (!waitingAckPackets.isEmpty()) {
          packet = (PacketInfo) waitingAckPackets.get();
        }
      }
      if (packet != null) {
        doAckPacketSend(packet);
        continue;
      } else {
        synchronized (waitingDataPackets) {
          if (!waitingDataPackets.isEmpty()) {
            packet = (PacketInfo) waitingDataPackets.get();
          }
        }
        doDataPacketSend(packet);
        // Sleep 25 ms +-
        Utils.sleep(25);
      }
    }

  }

}

/**
 * Attempts to transmit a packet using CSMA with ACK specific timings.
 *
 * @param packet the ack to be sent
 */
private void doAckPacketSend(PacketInfo packet) {
  boolean enabled = core.isEnabled();
  if (!enabled) {
    core.setEnabled(true);
  }
  boolean success = doSend(packet.packet);
```

```
      core.setEnabled(enabled);
      if (packet.listener != null) {
        packet.listener.sendDone(packet.packet, success);
      }
    }

    /**
     * Attempts to transmit a normal packet using CSMA.
     *
     * @param packet the packet to be sent
     */
    private void doDataPacketSend(PacketInfo packet) {
      boolean success = doSend(packet.packet);
      if (packet.listener != null) {
        packet.listener.sendDone(packet.packet, success);
      }
    }

    /**
     * Sends the packet.
     *
     * @param packet the packet to be sent
     * @param minBackoffExp minimum backoff exponent
     * @param maxBackoffExp max. backoff exponent
     * @param maxRetries max nr of retries
     * @return true when succeeded, false otherwise
     */
    private boolean doSend(MHMACPacket packet) {
      boolean channelClear = false;
      try {
        core.sendPacket(packet.asPHYPacket());
        channelClear = true;

      } catch (ChannelBusyException e) {
        channelClear = false;
      }

      return channelClear;
    }

}
```

Apêndice I. MHMACPacketTransmissionListener.java

```
package be.ac.ua.pats.anymac.mhmac;

/**
 * This interface allows MAC components that wish to send a packet to be updated
 * once the packet has been sent.
 */
public interface MHMACPacketTransmissionListener {

   /**
    * Called by {@link MHMACPacketSender} when the specified packet has been sent.
    * @param packet the packet that has been sent
    * @param succes whether the send was successfull
    */
   public void sendDone(MHMACPacket packet, boolean succes);

}
```

Apêndice J. MHMACSettingsManager.java

```
package be.ac.ua.pats.anymac.mhmac;

import be.ac.ua.pats.anymac.hal.components.ChannelController;
import be.ac.ua.pats.anymac.hal.components.PowerController;
import be.ac.ua.pats.anymac.mac.AbstractMACSettingsManager;
import be.ac.ua.pats.anymac.mac.MACSetting;
import be.ac.ua.pats.anymac.mac.MACSettingsManager;
import be.ac.ua.pats.anymac.mac.components.SimpleChannelSetting;
import be.ac.ua.pats.anymac.mac.components.SimplePowerSetting;

/**
 * The {@link MACSettingsManager} for the MHMAC.
 */
public class MHMACSettingsManager extends AbstractMACSettingsManager {

   /**
    * Constructor.
    * @param power the HAL Power Controller
    * @param channel the HAL Channel Controller
    */
   MHMACSettingsManager(PowerController power, ChannelController channel) {
      super(new MACSetting[]{new SimpleChannelSetting(channel), new SimplePowerSetting(power)});
   }

}
```

Apêndice K. DataCollectorUtils.java

```
package org.zephyr.datacollector;

import com.sun.spot.peripheral.IPowerController;
import com.sun.spot.peripheral.ISleepManager;
import java.io.IOException;
import javax.microedition.io.Datagram;

class DataCollectorUtils {

  /**
   * Broadcast port on which sensor samples are transmited and received.
   */
  static final int DATA_SINK_PORT = 67;

  static final long SLEEP_TIME = 5000;

  private DataCollectorUtils() {
  }

  /**
   * Writes all the data to be transmited into the datagram.
   * @param dg the datagram
   * @param time the time at which the mesurement was taken.
   * @throws IOException if an error occurs writing to the datagram.
   */
  static void writeToDatagram(Datagram dg, String address, long time, IPowerController
powerControler,
      ISleepManager sleepManager) throws IOException {
      double temperature = powerControler.getTemperature();
      int vbatt = powerControler.getVbatt();

      // Package our identifier, timestamp and sensor readings into a radio datagram.
      dg.reset();
      dg.writeUTF(address);
      dg.writeLong(time);
      dg.writeDouble(temperature);
      dg.writeInt(vbatt);
      dg.writeLong(sleepManager.getTotalDeepSleepTime());
      dg.writeLong(sleepManager.getTotalShallowSleepTime());
      dg.writeLong(sleepManager.getUpTime());

      // Log info to console
//                    logInfo("Reading at " + time + " is:");
//                    logInfo("\tTemperature: " + temperature);
//                    logInfo("\tBattery supply voltage: " + vbatt);
//                    logInfo("\tDeep sleep time: " + sleepManager.getTotalDeepSleepTime());
//                    logInfo("\tShallow sleep time: " + sleepManager.getTotalShallowSleepTime());
//                    logInfo("\tTotal time: " + sleepManager.getUpTime());
  }

  /**
   * Writes all the data to be transmited into the datagram.
   * @param dg the datagram
   * @param time the time at which the mesurement was taken.
```

```
   * @throws IOException if an error occurs writing to the datagram.
   */
  static Data readFromDatagram(Datagram dg) throws IOException {
    // Read all data from the datagram.
    String address = dg.readUTF();
    long time = dg.readLong();
    double temperature = dg.readDouble();
    int vbatt = dg.readInt();
    long totalDeepSleepTime = dg.readLong();
    long totalShallowSleepTime = dg.readLong();
    long upTime = dg.readLong();

    // Log info to console
//                  logInfo("Received data from " + address + " at " + time + ":");
//                  logInfo("\tTemperature: " + temperature);
//                  logInfo("\tBattery supply voltage: " + vbatt);
//                  logInfo("\tDeep sleep time: " + totalDeepSleepTime);
//                  logInfo("\tShallow sleep time: " + totalShallowSleepTime);
//                  logInfo("\tTotal time: " + upTime);

    return new Data(address, time, temperature, vbatt, totalDeepSleepTime, totalShallowSleepTime,
upTime);
  }

  /**
   * Logs a standard message.
   * @param msg the message to be logged.
   */
  static void logInfo(String msg) {
    // TODO: Add check based on property.
    System.out.println(msg);
  }

  /**
   * Logs an error message.
   * @param msg the message to be logged.
   */
  static void logError(String msg) {
    // TODO: Add check based on property.
    System.err.println(msg);
  }

  /**
   * Class to hold collected data.
   */
  static class Data {

    private final String address;

    private final long time;

    private final double temperature;

    private final int vbatt;

    private final long totalDeepSleepTime;

    private final long totalShallowSleepTime;

    private final long upTime;
```

```
    private Data(String address, long time, double temperature, int vbatt, long totalDeepSleepTime,
      long totalShallowSleepTime, long upTime) {
      this.address = address;
      this.time = time;
      this.temperature = temperature;
      this.vbatt = vbatt;
      this.totalDeepSleepTime = totalDeepSleepTime;
      this.totalShallowSleepTime = totalShallowSleepTime;
      this.upTime = upTime;
    }

    public String getDataValues() {
      return address + ";" + time + ";" + temperature + ";" + vbatt + ";" + totalDeepSleepTime + ";"
        + totalShallowSleepTime + ";" + upTime;
    }

    public static String getDataHeader() {
      return
"Address;Time;Temperature;Vbatt;TotalDeepSleepTime;TotalShallowSleepTime;UpTime";
    }

  }

}
```

Apêndice L. SPOTDataSender.java

```
package org.zephyr.datacollector;

import com.sun.spot.io.j2me.radiogram.RadiogramConnection;
import com.sun.spot.peripheral.IPowerController;
import com.sun.spot.peripheral.ISleepManager;
import com.sun.spot.peripheral.Spot;
import com.sun.spot.util.Utils;
import java.io.IOException;
import javax.microedition.io.Connector;
import javax.microedition.io.Datagram;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

/**
 * Simple app that gathers the SPOT info and sends it as a broadcast message.
 */
public class SPOTDataSender extends MIDlet {

  private String ourAddress;

  private IPowerController powerControler;

  private ISleepManager sleepManager;

  protected void destroyApp(boolean unconditional) throws MIDletStateChangeException {
  }

  protected void pauseApp() {
  }

  protected void startApp() throws MIDletStateChangeException {
    init();

    // Simulate deep sleep with USB cables connected.
    sleepManager.enableDiagnosticMode();

    RadiogramConnection rCon = null;
    Datagram dg = null;

    try {
//                          DataCollectorUtils.logInfo("Initializing connection.");
      // Open up a broadcast connection at the data sink port where the 'on Desktop' portion of this
demo is listening
      rCon = (RadiogramConnection) Connector.open("radiogram://broadcast:" +
DataCollectorUtils.DATA_SINK_PORT);
//                          DataCollectorUtils.logInfo("Creating datagram.");
      dg = rCon.newDatagram(rCon.getMaximumLength());
    } catch (Exception e) {
//                          DataCollectorUtils.logError("Caught " + e + " in connection initialization.");
      System.exit(1);
    }

    while (true) {
      long now = System.currentTimeMillis();
```

```
        try {
          // Get the current time and sensor readings
          DataCollectorUtils.writeToDatagram(dg, ourAddress, now, powerControler, sleepManager);
//                                      DataCollectorUtils.logInfo("Sending datagram.");
          rCon.send(dg);
        } catch (IOException e) {
//                                      DataCollectorUtils.logError("Caught " + e + " while collecting/sending
sensor sample.");
        }
        // Always go to sleep to conserve battery even if we got an error transmitting data.
        Utils.sleep(DataCollectorUtils.SLEEP_TIME);
      }
    }

    /**
     * Entry point for subclasses to insert simulated resources.
     */
    protected void init() {
      ourAddress = System.getProperty("IEEE_ADDRESS");
      powerControler = Spot.getInstance().getPowerController();
      sleepManager = Spot.getInstance().getSleepManager();
    }

}
```

Apêndice M. CollectDataHost.java

```
package org.zephyr.datacollector;

import com.sun.spot.io.j2me.radiogram.RadiogramConnection;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

import javax.microedition.io.Connector;
import javax.microedition.io.Datagram;
import org.zephyr.datacollector.DataCollectorUtils.Data;

public class CollectDataHost {

  private RadiogramConnection rCon = null;

  private Datagram dg = null;

  private Writer output = null;

  private void run() {
    try {
      setUp();
      collectData();
    } catch (Exception e) {
      DataCollectorUtils.logError("Terminating due to have caught " + e);
    } finally {
      tearDown();
    }
  }

  private void setUp() {
    try {
      rCon = (RadiogramConnection) Connector.open("radiogram://:" +
DataCollectorUtils.DATA_SINK_PORT);
      dg = rCon.newDatagram(rCon.getMaximumLength());

      // Setup the output file to store the received data
      output = new BufferedWriter(new FileWriter("outputData/data_" + System.currentTimeMillis() +
".csv"));
      output.write(Data.getDataHeader() + "\n");
    } catch (IOException e) {
      DataCollectorUtils.logError("Caught " + e + " in connection setup.");
      System.exit(1);
    }
    DataCollectorUtils.logInfo("setUp completed successfully.");
  }

  public void collectData() {
    DataCollectorUtils.logInfo("Starting to collect data.");
    // Main data collection loop
    while (true) {
      try {
        // Read sensor sample received over the radio
//                                    dg.reset();
//                                    dg.setLength(rCon.getMaximumLength());
        rCon.receive(dg);
```

```
            Data data = DataCollectorUtils.readFromDatagram(dg);
            // Indicate that some data was received
            DataCollectorUtils.logInfo("*");
            writeData(data);
          } catch (Exception e) {
            DataCollectorUtils.logError("Caught " + e + " while reading sensor samples.");
          }
        }
      }

      private void writeData(Data data) throws IOException {
        output.write(data.getDataValues() + "\n");
        // Also print to the screen for backup
        DataCollectorUtils.logInfo(data.getDataValues());
        // Must flush every write or the data will not be saved.
        output.flush();
      }

      public void tearDown() {
        DataCollectorUtils.logInfo("Closing resources.");
        if (rCon != null) {
          try {
            rCon.close();
          } catch (Exception e) {
            DataCollectorUtils.logError("Caught " + e + " while closing connection.");
          }
        }
        if (output != null) {
          try {
            output.close();
          } catch (IOException e) {
            DataCollectorUtils.logError("Caught " + e + " while closing file.");
          }
        }
      }

      /**
       * Start up the host application.
       *
       * @param args any command line arguments
       */
      public static void main(String[] args) {
        CollectDataHost app = new CollectDataHost();
        app.run();
      }

    }
```